NEAT Documentation

Release 0.1

The NEAT Authors

Dec 14, 2017

Articles

1 Welcome to the NEAT tutorial 1

2 Optional arguments 9

3 Properties 11

4 Architecture 15

5 Callbacks 17

6 Error codes 21

7 neat_init_ctx 23

8 neat_free_ctx 25

9 neat_new_flow 27

10 neat_set_property 29

11 neat_get_property 31

12 neat_open 33

13 neat_accept 35

14 neat_read 37

15 neat_write 39

16 neat_shutdown 41

17 neat_close 43

18 neat_abort 45

19 neat_set_operations 47

20 neat_change_timeout 49

21 neat_set_primary_dest 51

22 neat_secure_identity 53

23 neat_set_checksum_coverage 55

24 neat_set_qos 57

25 neat_get_qos 59

26 neat_set_ecn 61

27 neat_start_event_loop 63

28 neat_stop_event_loop 65

29 neat_get_backend_fd 67

30 neat_get_backend_timeout 69

31 neat_get_event_loop 71

32 neat_get_stats 73

33 neat_getlpaddrs 75

34 neat_log_level 77

35 neat_log_file 79

36 Coding Style 81

CHAPTER 1

Welcome to the NEAT tutorial

1.1 What is NEAT?

NEAT is a library for networked applications, intended to replace existing socket APIs with a simpler, more flexible API. Additionally, NEAT enables endpoints to make better decisions as to how to utilize the available network resources and adapts based on the current condition of the network.

With NEAT, applications are able to specify the service they want from the transport layer. NEAT will determine which of the available protocols fit the requirements of the application and tune all the relevant parameters to ensure that the application gets the desired service from the transport layer, based on knowledge about the current state of the network when this information is available.

NEAT enables applications to be written in a protocol-agnostic way, thus allowing applications to be future-proof, leveraging new protocols as they become available, with minimal to no change. Further, NEAT will try to connect with different protocols if possible, making it able to gracefully fall back to another protocol if it turns out that the most optimal protocol is unavailable, for example, because of a middlebox such as a firewall. A connection in the NEAT API will only fail if all protocols satisfying the requirements of the application are unable to connect, or if no available protocol can satisfy the requirements of the application.

Most operating systems support the same protocols. However, the same protocol may often have a slightly different API on different operating systems. NEAT provides the same API on all supported operating systems, which is currently Linux, FreeBSD, OpenBSD, NetBSD, and OS X. The availability of a protocol depends on whether the protocol is supported by the OS or if NEAT is compiled with support for a user-space stack that implements the protocol.

1.2 Contexts and flows

The most important concept in the NEAT API is that of the flow. A flow is similar to a socket in the traditional Berkely Socket API. It is a bidirectional link used to communicate between two applications, on which data may be written to or read from. Further, just like a socket, a flow uses some transport layer protocol to communicate.

However, one important difference is that a flow is not as strictly tied to the underlying transport protocol in the same way a socket is. In fact, a flow may be created without even specifying which transport protocol to use. This is not possible with a socket.

The same applies to modifying options on sockets. Setting the same kind of option on two sockets with different protocols in the traditional socket API requires `setsockopt` calls with different protocol IDs, option names,

and sometimes even values with different units. The `setsockopt` calls also vary depending on what system you are on. This is not the case with NEAT. As long as the desired option is available for the protocol in use, the API for setting that option is the same for all protocols, and on all operating systems supported by NEAT.

A context is a common environment for multiple flows. Along with flows, it contains several services that are used by the flows internally in NEAT, such as a DNS resolver and a Happy Eyeballs implementation. Flows within a context are polled together. A flow may only belong to the context in which it is created, and it cannot be transferred to a different context. Most applications need only one context.

1.3 Properties

Different types of applications have different requirements and desires to the services provided by the transport layer. An application for real-time communication may require the communication to have properties such as low latency, high bandwidth, quality of service, and have less strict requirements with regards to reliable delivery. Losing a packet or bit errors may be less critical to these applications. A web browser, on the other hand, might require communication that is (partially) ordered and error-free. A BitTorrent application might only require the ability to send packets to some destination with a minimum amount of effort, and not at the expense of other applications with stricter bandwidth requirements.

With the traditional socket API, the application requirements dictate the choice of which protocol to use. With NEAT, this is not the case. NEAT enables applications to specify the properties of the communication instead of specifying which protocol to use. Some properties may be required; other properties may be desired, but not mandatory. Based on the properties, NEAT will determine which protocols can support the requirements of the application and the options to set for each protocol. It will try to establish a connection by trying each of them until one connection succeeds, known as Happy Eyeballing.

The ability to specify properties instead of protocols allows applications to take advantage of available protocols where possible. By Happy Eyeballing, NEAT ensures that applications are able to cope with different network configurations, and gracefully fall back to another protocol if necessary should the most desirable protocol not be available for whatever reason.

1.4 Asynchronous API

The NEAT API is asynchronous and non-blocking. Once the execution is transferred to NEAT, it will poll the sockets internally, and, when an event happens, execute the appropriate callback in the application. This creates a more natural way of programming communicating applications than with the traditional socket API.

The three most important callbacks in the NEAT API are `on_connected`, `on_readable` and `on_writable`, which may be set per flow. The `on_connected` callback will be executed once the flow has connected to a remote endpoint, or a flow has connected to a server listening for incoming connections. The `on_writable` and `on_readable` callbacks are executed once data may be written to or read from the flow without blocking.

1.5 A minimal server

To get started using the NEAT API, we will write a small server that will send `Hello, this is NEAT!` to any client that connects to it. Later, we will write a similar client, before modifying this server so that it works with the client.

We can summarize the functionality as follows:

- When a client connects, start writing when the flow is writable
- When a flow is writable, write `Hello, this is NEAT!` to it.
- When the flow has finished writing, close it.

Pay close attention to how easily this natural description can be implemented using the NEAT API.

Here are the includes that should be put on top of the file:

```
#include <neat.h>

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

We will start writing the main function of our server. The first thing we need to do is to declare a few variables:

```
    struct neat_ctx *ctx;
    struct neat_flow *flow;
    struct neat_flow_operations ops;
```

And initialize them:

```
    ctx  = neat_init_ctx();
    if (!ctx) {
        fprintf(stderr, "neat_init_ctx failed\n");
        return EXIT_FAILURE;
```

We are already familiar with the flow and the context. `neat_init_ctx` is used to initialize the context, and `neat_new_flow` creates a new flow withing the context. The `neat_flow_operations` struct is used to tell NEAT what to do when certain events occur. We will use that next to tell which function we want NEAT to call when a client connects:

```
    }

    flow = neat_new_flow(ctx);
    if (!flow) {
```

The function `on_connected` has not been defined yet, we will do that later. Now that we have told NEAT what to do with a connecting client, we are ready to accept incoming connections.

```
        fprintf(stderr, "neat_new_flow failed\n");
        return EXIT_FAILURE;
    }

    memset(&ops, 0, sizeof(ops));
```

This will instruct NEAT to start listening to incoming connections on port 5000. The flow passed to `neat_accept` is cloned for each accepted connection. The last two parameters are used for optional arguments. This example does not use them.

The last function call we will do in main will be the one that starts the show:

```

    ops.on_readable  = on_readable;
```

When this function is called, NEAT will start doing work behind the scenes. When called with the `NEAT_RUN_DEFAULT` parameter, this function will not return until all flows have closed and all events have been handled. It is also possible to run NEAT without having NEAT capture the main loop. Our final main function looks like this:

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

```
    ctx  = neat_init_ctx();
    if (!ctx) {
        fprintf(stderr, "neat_init_ctx failed\n");
        return EXIT_FAILURE;
    }

    flow = neat_new_flow(ctx);
    if (!flow) {
        fprintf(stderr, "neat_new_flow failed\n");
        return EXIT_FAILURE;
    }

    memset(&ops, 0, sizeof(ops));

    ops.on_readable  = on_readable;
    ops.on_connected = on_connected;
    neat_set_operations(ctx, flow, &ops);
```

We have now filled in the main function of our server application. It is time to start working on the callbacks that NEAT will use. The first callback we need is `on_connected`.

```
static neat_error_code
on_connected(struct neat_flow_operations *opCB)
```

From the functional description above, we know that we need to write to connecting clients when this becomes possible. The callback contains a parameter that is a pointer to a `neat_flow_operations` struct, which we can use to update the active callbacks of the flow. We set the `on_writable` callback so that we can start writing when the flow becomes writable:

```
    opCB->on_writable    = on_writable;
```

It is also good practice to set the `on_all_written` callback when setting the `on_writable` callback:

```
    opCB->on_all_written = on_all_written;
```

The change is applied by calling `neat_set_operations`, just as in the main function:

```
    neat_set_operations(opCB->ctx, opCB->flow, opCB);
```

Next, we write the `on_writable` callback:

```
on_writable(struct neat_flow_operations *opCB)
{
```

Here, we call the function that will send our message:

```
    neat_write(opCB->ctx, opCB->flow, message, 20, NULL, 0);
    opCB->on_writable = NULL;
    return NEAT_OK;
```

Here we specify the data to send and the length of the data. As with the `neat_accept` function, `neat_write` takes optional parameters. We do not need to set any optional parameters for this call either, so again we pass `NULL` and `0`.

The final callback we need to implement is the `on_all_written` callback:

```
static neat_error_code
on_all_written(struct neat_flow_operations *opCB)
```

Here, we call `neat_close` to close the flow:

```
    neat_close(opCB->ctx, opCB->flow);
```

This is the final piece of our server. You may now compile and run the server. You can use the tool `socat` to test it. The following output should be observed:

```
$ socat STDIO TCP:localhost:5000
Hello, this is NEAT!
$ socat STDIO SCTP:localhost:5000
Hello, this is NEAT!
```

You may find the complete source for the server here.

1.6 A minimal client

Next, we want to implement a client that will send the message `"Hi!"` after connecting to a server, and then receive a reply from the server. A fair amount of the code will be similar to the server we wrote above, so you may make a copy of the code for the server and use that as a starting point for the client.

We will make two additions and one change to the main function. First, since we are connecting to a server, we change the `neat_accept` call to `neat_open` instead:

```
    }

    flow = neat_new_flow(ctx);
    if (!flow) {
```

Next, we will specify a few properties for the flow:

```
static char *properties = "{\n\
    \"transport\": [\n\
        {\n\
            \"value\": \"SCTP\",\n\
            \"precedence\": 1\n\
        },\n\
        {\n\
            \"value\": \"TCP\",\n\
            \"precedence\": 1\n\
        }\n\
    ]\n\
}";
```

These properties will tell NEAT that it can select either SCTP or TCP as the transport protocol. The properties are applied with `neat_set_properties`, which may be done at any point between `neat_new_flow` and `neat_open`.

Finally, we add `neat_free_ctx` after `neat_start_event_loop`, so that NEAT may free any allocated resources and exit gracefully. The complete main function of the client will look like this:

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

    ctx  = neat_init_ctx();
    if (!ctx) {
        fprintf(stderr, "neat_init_ctx failed\n");
        return EXIT_FAILURE;
    }
```

```
    flow = neat_new_flow(ctx);
    if (!flow) {
        fprintf(stderr, "neat_new_flow failed\n");
        return EXIT_FAILURE;
    }

    memset(&ops, 0, sizeof(ops));

    ops.on_connected = on_connected;
    ops.on_close = on_close;
    neat_set_operations(ctx, flow, &ops);

    if (neat_set_property(ctx, flow, properties) != NEAT_OK) {
        fprintf(stderr, "neat_set_property failed\n");
        return EXIT_FAILURE;
```

Leave the `on_connected` callback similar to the server.

We change the `on_writable` callback to send `"Hi!"` instead:

```
static neat_error_code
on_writable(struct neat_flow_operations *ops)
{
    const unsigned char message[] = "Hi!";
    neat_write(ops->ctx, ops->flow, message, 3, NULL, 0);
    return NEAT_OK;
}
```

The `on_all_written` callback should not close the flow, but instead stop writing and set the `on_readable` callback:

```
static neat_error_code
on_all_written(struct neat_flow_operations *ops)
{
    ops->on_readable = on_readable;
    ops->on_writable = NULL;
    neat_set_operations(ops->ctx, ops->flow, ops);
    return NEAT_OK;
}
```

Finally, we will write an `on_readable` callback for the client. We allocate some space on the stack to store the received data, and use a variable to store the length of the received message. If the `neat_read` call returns successfully, we print the message. Finally, we stop the internal event loop in NEAT, which will eventually cause the call to `neat_start_event_loop` in the main function to return. The `on_readable` callback should look like this:

```
static neat_error_code
on_readable(struct neat_flow_operations *ops)
{
    uint32_t bytes_read = 0;
    unsigned char buffer[32];

    if (neat_read(ops->ctx, ops->flow, buffer, 31, &bytes_read, NULL, 0) == NEAT_
→OK) {
        buffer[bytes_read] = 0;
        fprintf(stdout, "Read %u bytes:\n%s", bytes_read, buffer);
    }

    neat_close(ops->ctx, ops->flow);
    return NEAT_OK;
}
```

And there we have our finished client! You can test it with `socat`:

```
$ socat TCP-LISTEN:5000 STDIO
```

When you run the client, you should see `Hi!` show up in the output from socat. You can type a short message followed by pressing return, and it should show up in the output on the client.

You may find the complete source for the client here.

1.7 Tying the client and server together

A few small changes are required on the server to make the client and server work together. In the `on_connected` callback, the server should set the `on_readable` callback instead of the `on_writable` callback. An `on_readable` callback should be added and read the incoming message from the client, and set the `on_writable` callback.

The callbacks for the updated server is as follows:

```
static neat_error_code
on_readable(struct neat_flow_operations *ops)
{
    uint32_t bytes_read = 0;
    unsigned char buffer[32];

    if (neat_read(ops->ctx, ops->flow, buffer, 31, &bytes_read, NULL, 0) == NEAT_
↪OK) {
        buffer[bytes_read] = 0;
        fprintf(stdout, "Read %u bytes:\n%s\n", bytes_read, buffer);
    }

    ops->on_readable = NULL;
    ops->on_writable = on_writable;
    ops->on_all_written = on_all_written;
    neat_set_operations(ops->ctx, ops->flow, ops);

    return NEAT_OK;
}

static neat_error_code
on_writable(struct neat_flow_operations *ops)
{
    const unsigned char message[] = "Hello, this is NEAT!";
    neat_write(ops->ctx, ops->flow, message, 20, NULL, 0);

    ops->on_writable = NULL;
    ops->on_readable = NULL;
    ops->on_all_written = on_all_written;
    neat_set_operations(ops->ctx, ops->flow, ops);

    return NEAT_OK;
}

static neat_error_code
on_all_written(struct neat_flow_operations *ops)
{
    ops->on_readable = NULL;
    ops->on_writable = NULL;
    ops->on_all_written = NULL;
    neat_set_operations(ops->ctx, ops->flow, ops);

    neat_close(ops->ctx, ops->flow);
```

```
    return NEAT_OK;
}

static neat_error_code
on_connected(struct neat_flow_operations *ops)
{
    ops->on_readable = on_readable;
    neat_set_operations(ops->ctx, ops->flow, ops);

    return NEAT_OK;
}
```

You may find the complete source for the updated server here. A minimal CMakeLists.txt for cmake can be found here.

CHAPTER 2

Optional arguments

Some of the functions in the NEAT API, such as `neat_open`, `neat_read` and `neat_write`, take optional arguments. These are sometimes used to pass optional arguments to functions, and sometimes used to return additional values from the function. Optional arguments are passed as an array of the struct `neat_tlv` and an integer specifying the length of this array. `neat_tlv` is defined as follows:

```
struct neat_tlv {
    neat_tlv_tag  tag;
    neat_tlv_type type;

    union {
        int   integer;
        char *string;
        float real;
    } value;
};
```

An optional argument takes the form of a tag name, a type, and a value of either a string, integer or a floating point number. The tag specifies which optional argument the value belongs to, and the type asserts the type of the value passed as this argument. An error will be raised of the type is different than what the function expects.

You may either work with this struct directly, or you may use the preprocessor macros explained later in this document.

2.1 Specifying no optional arguments

To specify no optional arguments, simply pass `NULL` as the `optargs` parameter and `0` as the `opt_count` parameter of the function.

2.2 Optional argument macros

- **NEAT_OPTARGS_DECLARE(max)** - Declare the necessary variables to use the rest of these macros. Allocates (on the stack) an array of length *max* and an integer for storing the number of optional arguments specified. `NEAT_OPTARGS_MAX` may be used as the default array size.

- **NEAT_OPTARGS_INIT()** - Initializes the variables declared by `NEAT_OPTARGS_DECLARE`. May also be used to reset the (number of) optional arguments back to 0.
- **NEAT_OPTARG_INT(tag,value)** - Specify the tag and the value of an optional argument that takes an integer.
- **NEAT_OPTARG_FLOAT(tag,value)** - Specify the tag and the value of an optional argument that takes a floating point number.
- **NEAT_OPTARG_STRING(tag,value)** - Specify the tag and the value of an optional argument that takes a string.
- **NEAT_OPTARGS** - Represents the array of optional arguments. Specify this macro as the `optarg` parameter.
- **NEAT_OPTARG_COUNT** - Stores the number of the optional arguments specified so far with `NEAT_OPTARG_INT`, `NEAT_OPTARG_FLOAT` or `NEAT_OPTARG_STRING`. This count is reset by `NEAT_OPTARGS_INIT()`. Specify this macro as the `opt_count` argument.

2.3 Optional argument tags

- **NEAT_TAG_STREAM_ID** (integer) - Specifies the ID of the stream which the data should be written to, or which stream the data was read from.
- **NEAT_TAG_STREAM_COUNT** (integer) - Specifies the number of stream to create. Only used with protocols that support multiple streams.
- **NEAT_TAG_FLOW_GROUP** (int) - Specifies the flow group this flow belongs to.
- **NEAT_TAG_PRIORITY** (float) - Specifies the priority of this flow relative to other flows in the flow group.
- **NEAT_TAG_CC_ALGORITHM** (string) - Speficies the name of the (TCP) congestion control algorithm that will be used by this flow. A system default will be used if the specified algorithm is not available.

Currently unused tags:

- **NEAT_TAG_LOCAL_NAME**
- **NEAT_TAG_SERVICE_NAME**
- **NEAT_TAG_CONTEXT**
- **NEAT_TAG_PARTIAL_RELIABILITY_METHOD**
- **NEAT_TAG_PARTIAL_RELIABILITY_VALUE**
- **NEAT_TAG_PARTIAL_MESSAGE_RECEIVED**
- **NEAT_TAG_PARTIAL_SEQNUM**
- **NEAT_TAG_UNORDERED**
- **NEAT_TAG_UNORDERED_SEQNUM**
- **NEAT_TAG_DESTINATION_IP_ADDRESS**

2.4 Examples

```
NEAT_OPTARGS_DECLARE(NEAT_OPTARGS_MAX);
NEAT_OPTARGS_INIT();
NEAT_OPTARG_INT(NEAT_TAG_STREAM_COUNT, 5);
neat_open(ctx, flow, "127.0.0.1", 8000, NEAT_OPTARGS, NEAT_OPTARGS_COUNT);
```

CHAPTER 3

Properties

A property in NEAT may either express a requirement or it may express a desire from the application with regards to the service provided by the transport layer.

Properties are represented as JSON objects. A set of properties may be contained within one JSON object. Below is an example of a JSON object with a single property:

```
{
    "property_name": {
        value: "property_value",
        precedence: 1
    }
}
```

Note that all examples of properties will be specified inside a JSON object.

Properties have a name, a value, and a precedence. A string is always used for the name of a property. The value of a property may be either a boolean, a string, an integer, a floating point number, an array, or an interval. Each property expects only one specific type.

The properties are sent to the Policy Manager (if present), which will return a list containing a list of candidates, which is ordered by how good the candidate matches the request from the application. Each candidate specifies a given setting for each property. NEAT will use the properties specified in each candidate when trying to set up a new connection.

Some properties are set by NEAT based on parameters to function calls. Other properties must be set manually with the `neat_set_property` function.

3.1 Application property reference

These are properties that may be set by the application.

3.1.1 transport

Type: Array

Specifies an array of transport protocols that may be used. An application may specify either one protocol with precedence 2, or multiple protocols with precedence 1.

Note: May not be queried with `neat_get_property` before execution of the `on_connected` callback. When querying this property, the returned value is a string describing the actual transport in use.

Note: Applications should avoid specifying the protocol(s) to use directly, and instead rely on the Policy Manager to make a decision on what protocol(s) to use based on other properties. The `transport` property should only be used for systems without a Policy Manager, or if the choice of transport protocol is strictly mandated by the application protocol.

Example 1: Multiple protocols

```
{
    "transport":
    {
        "value": [ "SCTP", "TCP" ]
        "precedence": 1
    }
}
```

Example 2: One protocol

```
{
    "transport":
    {
        "value": "UDP",
        "precedence": 2
    }
}
```

Available protocols:

- SCTP
- SCTP/UDP (SCTP tunneled over UDP)
- TCP
- UDP
- UDP-Lite

3.1.2 security

Type: Boolean

Specifies whether the connection should be encrypted or not. With precedence 2, NEAT will only report the connection as established if it was able to connect and the (D)TLS handshake succeeds. With precedence 1, NEAT may still attempt to establish an unencrypted connection.

3.2 Inferred properties

These are properties that are inferred during connection setup and subsequently sent to the Policy Manager. Applications should not set these directly.

3.2.1 interfaces

Type: Array

Specifies a list of available interfaces that may be used for connections. The Policy Manager may not use all interfaces in this list.

This property is inferred during the `neat_open` call. Do not set this property manually.

3.2.2 domain_name

Type: String

Specifies the name of the remote endpoint to connect to with the `neat_open` call.

This property is inferred from the `name` parameter of `neat_open` call. Do not set this property manually.

3.2.3 port

Type: Integer

This property is inferred from the `neat_open` and `neat_accept` calls. Do not set this property manually.

CHAPTER 4

Architecture

4.1 Application

The application in NEAT terminology is the software using the transport layer to communicate with a remote endpoint.

4.2 NEAT API

The NEAT API is the public interface which applications may use to implement communication over the transport layer.

4.3 NEAT Core

The NEAT Core ties all the various components of the NEAT framework together, and handles communication over the transport layer once the connection has been established.

4.4 Connection setup in NEAT

The following is a description of how a connection is set up in NEAT:

1. The application specifies properties of the communication with `neat_set_property`.
2. The application calls `neat_open` in the NEAT API.
3. The NEAT Core sends application properties and inferred properties to the Policy Manager.
4. The Policy Manager replies with an initial set of candidates eligible for name resolution (e.g. DNS lookup).
5. The NEAT Core initiates one or more name resolution requests to the Name Resolver.
6. The Name Resolver replies with resolved addresses, and the NEAT Core inserts these into each candidate.
7. The NEAT Core makes a second call to the Policy Manager.

8. The Policy Manager returns a list of suitable candidates that the NEAT Core should use to establish a connection. A candidate consists of the following:

   - Transport protocol
   - Interface
   - Port
   - Local address
   - Remote address
   - Priority
   - Application properties

   If one or more of the application properties are specified as desired (precedence 1), multiple candidates *may* be generated with different settings for that property.

9. The NEAT Core generates a list of Happy Eyeball candidates and initiates the Happy Eyeballs algorithm.
10. The Happy Eyeballs module tries to connect each candidate in turn. The delay between each candidate is determined from the priority of the candidate. A lower value implies a higher priority. The connection may be handled by either the operating system using its own implementation of the protocol, or using a userspace implementation of the protocol.
11. The first connection that connects successfully and meets all required properties set by the application is returned to the NEAT Core.
12. NEAT starts polling the socket internally, and reports back to the application that a connection has been established using the `on_connected` callback if it has been specified using `neat_set_operations`.
13. NEAT will report that the flow is readable or writable if the respective `on_readable` or `on_writable` callbacks have been specified with `neat_set_operations`.
14. The application closes the flow with `neat_close`.

CHAPTER 5

Callbacks

Callbacks are used in NEAT to signal events to the application. They are used to inform the application when a flow is readable, writable, or an error has occurred.

Most callbacks have the following syntax:

```
neat_error_code
on_event_name(neat_flow_operations *ops)
{
    return NEAT_OK; // or some error code
}
```

The struct `neat_flow_operations` is defined as follows:

```
struct neat_flow_operations
{
    void *userData;

    neat_error_code status;
    int stream_id;
    struct neat_ctx *ctx;
    struct neat_flow *flow;

    neat_flow_operations_fx on_connected;
    neat_flow_operations_fx on_error;
    neat_flow_operations_fx on_readable;
    neat_flow_operations_fx on_writable;
    neat_flow_operations_fx on_all_written;
    neat_flow_operations_fx on_network_status_changed;
    neat_flow_operations_fx on_aborted;
    neat_flow_operations_fx on_timeout;
    neat_flow_operations_fx on_close;
    neat_cb_send_failure_t on_send_failure;
    neat_cb_flow_slowdown_t on_slowdown;
    neat_cb_flow_rate_hint_t on_rate_hint;
};
```

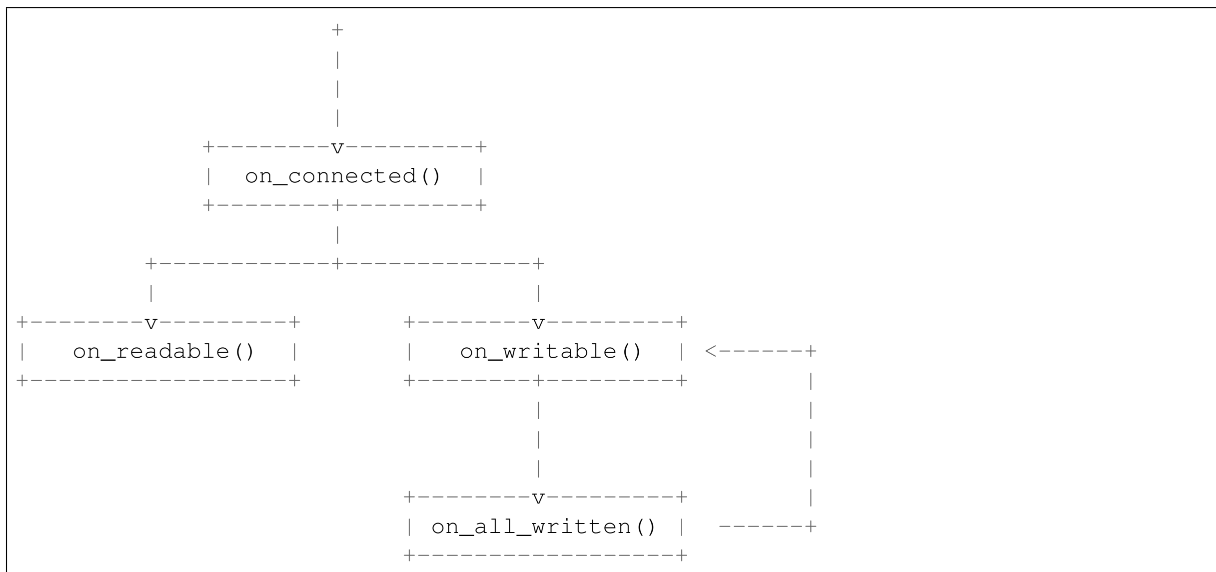
- **userData**: Applications may freely store a pointer in this field.
- **status**: Reports any errors associated with the flow.
- **stream_id**: For flows that use explicit multistreaming. Specifies which stream the event is related to, if any.

- **ctx**: Pointer to the context the flow belongs to.
- **flow**: Pointer to the flow on which the event happened.

Callbacks are set by assigning the function pointer to the struct passed to the callback and then calling `neat_set_operations`. A `NULL` pointer may be used to indicate that the callback should no longer be called.

5.1 Example callback flow

For most applications it will be sufficient to use the following callback flow:



See the *tutorial* for more details.

5.2 Callback reference

5.2.1 on_connected

Called whenever an outgoing connection has been established with `neat_open`, or an incoming connection has been established with `neat_accept`.

5.2.2 on_error

Called whenever an error occurs when processing the flow. Errors are considered critical.

5.2.3 on_readable

Called whenever the flow can be read from without blocking. NEAT does not permit blocking reads.

5.2.4 on_writable

Called whenever the flow can be written to without blocking. NEAT does not permit blocking writes.

5.2.5 on_all_written

Called when all previous data sent with `neat_write` has been completely written. Does not signal that the flow is writable. Applications may use this callback to re-enable the `on_writable` callback.

5.2.6 on_network_status_changed

Inform application that something has happened in the network. This also includes flow endpoints going up, which will subsequently trigger `on_connected` if that callback is set.

Only available when using SCTP.

5.2.7 on_aborted

Called when the remote end aborts the flow. Available for flows using TCP or SCTP.

5.2.8 on_timeout

Called if sent data is not acknowledged within the time specified with `neat_change_timeout`.

Currently only available for TCP on Linux.

5.2.9 on_close

Called when the graceful connection shutdown has completed.

Only available when using SCTP or TCP. Note that when using TCP, this callback is called when the `close()` system call is made, as TCP implementations currently does not provide any more accurate way of signalling this.

5.2.10 on_send_failure

Defined as:

```
void
on_send_failure(struct neat_flow_operations *flowops, int context, const unsigned␣
↪char *unsent)
{
}
```

Called to inform the application that the returned message `unsent` could not be transmitted. The failure reason as reported by the transport protocol is returned in the standard status code, as an abstracted NEAT error code. If the message was tagged with a context number, it is returned in context.

Only available for SCTP. Flows using TCP may use timeouts instead.

5.2.11 on_slowdown

Not currently implemented.

Defined as:

```
void
on_slowdown(struct neat_flow_operations *ops, int ecn, uint32_t rate)
{
}
```

Inform the application that the flow has experienced congestion and that the sending rate should be lowered. If `rate` is non-zero, it is an estimate of the new maximum sending rate. `ecn` is a boolean indicating whether this notification was triggered by an ECN mark.

5.2.12 on_rate_hint

Not currently implemented.

Defined as:

```
void
on_rate_hint(struct neat_flow_operations *ops, uint32_t new_rate)
{
}
```

Called to inform the application that it may increase its sending rate. If `new_rate` is non-zero, it is an estimate of the maximum sending rate.

CHAPTER 6

Error codes

6.1 NEAT_OK

Signals that no error has occurred. Equals to `0`.

6.2 NEAT_ERROR_WOULD_BLOCK

Signals that the operation could not be performed because it would block the process. NEAT does not permit blocking operations.

6.3 NEAT_ERROR_BAD_ARGUMENT

Signals that one or more arguments given to the function was invalid or incorrect. This also includes optional arguments.

6.4 NEAT_ERROR_IO

Signals that an internal I/O operation in NEAT has failed.

6.5 NEAT_ERROR_DNS

Signals that there was an error performing DNS resolution.

6.6 NEAT_ERROR_INTERNAL

Signals that there was an error internally in NEAT.

6.7 NEAT_ERROR_SECURITY

Signals that there was an error setting up an encrypted flow.

6.8 NEAT_ERROR_UNABLE

Signals that NEAT is not able to perform the requested operation.

6.9 NEAT_ERROR_MESSAGE_TOO_BIG

Signals that the provided buffer space is not sufficient for the received message.

6.10 NEAT_ERROR_REMOTE

Signals that there was an error on the remote endpoint.

6.11 NEAT_ERROR_OUT_OF_MEMORY

Signals that NEAT is not able to allocate enough memory to complete the requested operation.

CHAPTER 7

neat_init_ctx

Summary.

7.1 Syntax

```
struct neat_ctx *neat_init_ctx();
```

7.2 Parameters

None.

7.3 Return values

- Returns a pointer to a newly allocated and initialized NEAT context.
- Returns `NULL` if a context could not be allocated.

7.4 Remarks

None.

7.5 Examples

None.

7.6 See also

- *neat_free_ctx*
- *neat_new_flow*

CHAPTER 8

neat_free_ctx

Summary.

8.1 Syntax

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

8.2 Parameters

- **ctx**: Pointer to the NEAT context to free.

8.3 Return values

None.

8.4 Remarks

If there are any flows still kept in this context, those will be freed and closed as part of this operation.

8.5 Examples

None.

8.6 See also

- neat_close

- *neat_init_ctx*
- *neat_new_flow*

CHAPTER 9

neat_new_flow

Allocate and initialize a new NEAT flow.

9.1 Syntax

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

9.2 Parameters

- **ctx**: Pointer to a NEAT context.

9.3 Return values

Returns a pointer to a new flow. Returns `NULL` on error.

9.4 Remarks

None.

9.5 Examples

None.

9.6 See also

- *neat_open*
- *neat_close*

CHAPTER 10

neat_set_property

Set the properties of a NEAT flow.

10.1 Syntax

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

10.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **properties**: Pointer to a JSON-encoded string containing the flow properties.

10.3 Return values

- Returns `NEAT_OK` if the properties were set successfully.
- Returns `NEAT_ERROR_BAD_ARGUMENT` if the JSON-encoded string is malformed.

10.4 Remarks

Properties are applied when a flow connects.

10.5 Examples

None.

10.6 See also

None.

CHAPTER 11

neat_get_property

Query the properties of a flow. Returns value only, not precedence.

11.1 Syntax

```
neat_error_code neat_get_property(struct neat_ctx *ctx,
                                  struct neat_flow *flow,
                                  const char *name,
                                  void *ptr,
                                  size_t *size);
```

11.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **name**: Name of the property to query.
- **ptr**: Pointer to a buffer where the property value may be stored.
- **size**: Pointer to an integer containing the size of the buffer pointed to by `ptr`. Updated to contain the size of the property upon return.

11.3 Return values

- Returns `NEAT_OK` if the property existed and there was sufficient buffer space available. The `size` parameter is updated to contain the actual size.
- Returns `NEAT_ERROR_MESSAGE_TOO_BIG` if there was not sufficient buffer space. The `size` parameter is updated to contain the required buffer size.
- Returns `NEAT_ERROR_UNABLE` if the property does not exist.

11.4 Remarks

Applications may pass `0` as the `size` parameter to query the size of the property.

11.5 Examples

```
size_t bufsize = 0;
char buffer = NULL;

if (neat_get_property(ctx, flow, "transport", buffer, &bufsize) == NEAT_ERROR_
↪MESSAGE_TOO_BIG) {
    buffer = malloc(bufsize);
    if (buffer && neat_get_property(ctx, flow, "transport", buffer, &bufsize)␣
↪== NEAT_OK) {
        printf("Transport: %s\n", buffer);
    }
    if (buffer)
        free(buffer);
} else {
    printf("\tTransport: Error: Could not find property \"transport\"\n");
}
```

11.6 See also

- *Properties*
- *neat_set_property*

CHAPTER 12

neat_open

Open a neat flow and connect it to a given remote name and port.

12.1 Syntax

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

12.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **name**: The remote name to connect to.
- **port**: The remote port to connect to.
- **optional**: An array containing optional parameters.
- **opt_count**: The length of the array containing optional parameters.

12.3 Optional parameters

- **NEAT_TAG_STREAM_COUNT** (integer): The number of streams to open, for protocols that supports multistreaming. Note that NEAT may automatically make use of multi-streaming for multiple NEAT flows between the same endpoints when this parameter is not used.
- **NEAT_TAG_FLOW_GROUP** (integer): The group ID that this flow belongs to. For use with coupled congestion control.

- **NEAT_TAG_PRIORITY** (float): The priority of this flow relative to the other flows. Must be between 0.1 and 1.0.
- **NEAT_TAG_CC_ALGORITHM** (string): The congestion control algorithm to use for this flow.

12.4 Return values

- Returns `NEAT_OK` if the flow opened successfully.
- Returns `NEAT_ERROR_OUT_OF_MEMORY` if the function was unable to allocate enough memory.

12.5 Remarks

Callbacks can be specified with `neat_set_operations`. The `on_connected` callback will be invoked if the connection established successfully. The `on_error` callback will be invoked if NEAT is unable to connect to the remote endpoint.

12.6 Examples

```
neat_open(ctx, flow, "bsd10.fh-muenster.de", 80, NULL, 0);
```

12.7 See also

- *neat_read*
- *Optional arguments*

CHAPTER 13

neat_accept

Listen to incoming connections on a given port on one or more protocols.

13.1 Syntax

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

13.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **port**: The local port to listen for incoming connections on.
- **optional**: An array containing optional parameters.
- **opt_count**: The length of the array containing optional parameters.

13.3 Optional parameters

- **NEAT_TAG_STREAM_COUNT** (integer): The number of streams to accept, for protocols that supports multistreaming.

13.4 Return values

- Returns `NEAT_OK` if NEAT is listening for incoming connections on at least one protocol.

- Returns `NEAT_ERROR_UNABLE` if there is no appropriate protocol available for the flow properties that was specified.
- Returns `NEAT_ERROR_BAD_ARGUMENT` if **flow** is pointing to a flow that is already opened or listening for incoming connections.
- Returns `NEAT_ERROR_BAD_ARGUMENT` if **NEAT_TAG_STREAM_COUNT** is less than 1.
- Returns `NEAT_ERROR_OUT_OF_MEMORY` if the function was unable to allocate enough memory.

13.5 Remarks

Callbacks can be specified with `neat_set_operations`. The `on_connected` callback will be invoked if the connection established successfully. The `on_error` callback will be invoked if NEAT is unable to connect to the remote endpoint.

Which protocols to listen to is determined by the flow properties.

13.6 Examples

```
neat_accept(ctx, flow, 8080, NULL, 0);
```

13.7 See also

- *neat_open*
- *Optional arguments*

CHAPTER 14

neat_read

Read data from a neat flow.

Should only be called from within the `on_readable` callback specified with `neat_set_operations`.

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

14.1 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **buffer**: Pointer to a buffer where read data may be stored.
- **amount**: The size of the buffer pointed to by **buffer**.
- **actual_amount**: The amount of data actually read from the transport layer.
- **optional**: An array containing optional parameters.
- **opt_count**: The length of the array containing optional parameters.

14.2 Optional parameters

This function uses optional parameters for some return values.

- **NEAT_TAG_STREAM_ID** (integer): The ID of the stream will be written to this parameter.

14.3 Return values

- Returns `NEAT_OK` if data was successfully read from the transport layer.
- Returns `NEAT_ERROR_WOULD_BLOCK` if this call would block.
- Returns `NEAT_ERROR_MESSAGE_TOO_BIG` if the **buffer** is not sufficiently large. This is only returned for protocols that are message based, such as UDP, UDP-Lite and SCTP.
- Returns `NEAT_ERROR_IO` if the connection is reset.

14.4 Remarks

This function should only be called from within the `on_readable` callback specified with `neat_set_operations`, as this is the only way to guarantee that the call will not block. NEAT does not permit a blocking read operation.

The **actual_amount** value is set to 0 when the remote side has closed the connection.

14.5 Examples

None.

14.6 See also

- *neat_write*
- *Optional arguments*

CHAPTER 15

neat_write

Write data to a neat flow. Should only be called from within the on_writable callback specified with neat_set_operations.

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

15.1 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **buffer**: Pointer to a buffer containing the data to be written.
- **amount**: The size of the buffer pointed to by **buffer**.
- **optional**: An array containing optional parameters.
- **opt_count**: The length of the array containing optional parameters.

15.2 Optional parameters

- **NEAT_TAG_STREAM_ID** (integer): The ID of the stream the data will be written to.

15.3 Return values

- Returns NEAT_OK if data was successfully written to the transport layer.
- Returns NEAT_ERROR_BAD_ARGUMENT if the specified stream ID is negative.
- Returns NEAT_ERROR_OUT_OF_MEMORY if NEAT is unable to allocate memory.

- Returns `NEAT_ERROR_WOULD_BLOCK` if this call would block.
- Returns `NEAT_ERROR_IO` if an I/O operation failed.

15.4 Remarks

This function should only be called from within the `on_writable` callback specified with `neat_set_operations`, as this is the only way to guarantee that the call will not block. NEAT does not permit a blocking write operation.

Invalid stream IDs are silently ignored.

15.5 Examples

None.

15.6 See also

- *neat_read*
- *Optional arguments*

CHAPTER 16

neat_shutdown

Initiate a graceful shutdown of this flow.

- the receive buffer can still be read and `on_readable` gets fired like in normal operation
- receiving **new** data from the peer **may** fail
- all data in the *send buffer* will be transmitted
- `neat_write` will fail and `on_writable` will not be called

If the peer also has closed the connection, the `on_close` callback gets fired.

16.1 Syntax

```
neat_error_code neat_shutdown(struct neat_ctx *ctx,
                              struct neat_flow *flow);
```

16.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to the NEAT flow to be shut down.

16.3 Return values

- Returns `NEAT_OK` if the flow was shut down successfully.
- Returns `NEAT_ERROR_IO` if NEAT was unable to shut the flow down successfully.

16.4 Remarks

None.

16.5 Examples

None.

16.6 See also

- *neat_close*

CHAPTER 17

neat_close

Close this flow and free all associated data. If the peer still has data to send, it cannot be received anymore after this call. Data buffered by the NEAT layer which has not given to the network layer yet will be discarded.

17.1 Syntax

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

17.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to the NEAT flow to be closed.

17.3 Return values

- Returns `NEAT_OK`.

17.4 Remarks

None.

17.5 Examples

None.

17.6 See also

- *neat_shutdown*

CHAPTER 18

neat_abort

Abort this flow and free all associated data.

18.1 Syntax

```
neat_error_code neat_abort(struct neat_ctx *ctx,
                           struct neat_flow *flow);
```

18.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to the NEAT flow to be aborted.

18.3 Return values

- Returns `NEAT_OK`.

18.4 Remarks

Calls `neat_close` internally.

18.5 Examples

None.

18.6 See also

- *neat_shutdown*
- *neat_close*

CHAPTER 19

neat_set_operations

Summary.

19.1 Syntax

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

19.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **ops**: Pointer to a struct that defines the operations/callbacks for this flow.

19.3 Return values

- Returns `NEAT_OK`.

19.4 Remarks

`struct neat_flow_operations` is defined as follows:

```
struct neat_flow_operations
{
    void *userData;

    neat_error_code status;
    int stream_id;
```

```
    neat_flow_operations_fx on_connected;
    neat_flow_operations_fx on_error;
    neat_flow_operations_fx on_readable;
    neat_flow_operations_fx on_writable;
    neat_flow_operations_fx on_all_written;
    neat_flow_operations_fx on_network_status_changed;
    neat_flow_operations_fx on_aborted;
    neat_flow_operations_fx on_timeout;
    neat_flow_operations_fx on_close;
    neat_cb_send_failure_t on_send_failure;
    neat_cb_flow_slowdown_t on_slowdown;
    neat_cb_flow_rate_hint_t on_rate_hint;

    struct neat_ctx *ctx;
    struct neat_flow *flow;
};
```

The information in the **ops** struct will be copied by NEAT.

19.5 Examples

```
struct neat_flow_operations ops;
ops.on_readable = on_readable;
ops.on_writable = on_writable;
neat_set_operations(ctx, flow, ops);
```

19.6 See also

None.

CHAPTER 20

neat_change_timeout

Change the timeout of the flow. Data that is sent may remain un-acked for up to a given number of seconds before the connection is terminated and a timeout is reported to the application.

20.1 Syntax

```
neat_error_code
neat_change_timeout(struct neat_ctx *ctx, struct neat_flow *flow,
                    unsigned int seconds);
```

20.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **seconds**: The number of seconds after which un-acked data will cause a timeout to be reported.

20.3 Return values

- Returns `NEAT_OK` if the timeout was successfully changed.
- Returns `NEAT_ERROR_UNABLE` if attempting to use this function on a system other than Linux, or on flow that is not using TCP.
- Returns `NEAT_ERROR_BAD_ARGUMENT` if the timeout value is too large or if the specified flow is not opened.
- Returns `NEAT_ERROR_IO` if NEAT was unable to set the timeout.

20.4 Remarks

Only available on Linux for flows using TCP.

20.5 Examples

None.

20.6 See also

None.

CHAPTER 21

neat_set_primary_dest

For multihomed flows, set the primary destination address.

21.1 Syntax

```
neat_error_code neat_set_primary_dest(struct neat_ctx *ctx,
                                      struct neat_flow *flow,
                                      const char* address);
```

21.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **address**: The remote address to use as the primary destination address.

21.3 Return values

- Returns `NEAT_OK` if the primary destination address was set successfully.
- Returns `NEAT_ERROR_UNABLE` if the flow is not using SCTP as the transport protocol.
- Returns `NEAT_ERROR_BAD_ARGUMENT` if the provided address is not a literal IP address.

21.4 Remarks

Currently only available for SCTP.

21.5 Examples

None.

21.6 See also

None.

CHAPTER 22

neat_secure_identity

Specify a certificate and key to use for secure connections.

22.1 Syntax

```
neat_error_code neat_secure_identity(
    struct neat_ctx *ctx,
    struct neat_flow *flow,
    const char *filename);
```

22.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **filename**: Path to the PEM file containing the certificate and key.

22.3 Return values

- Returns `NEAT_OK`.

22.4 Remarks

None.

22.5 Examples

None.

22.6 See also

None.

CHAPTER 23

neat_set_checksum_coverage

Set the checksum coverage for messages sent or received on this flow.

23.1 Syntax

```
neat_error_code neat_set_checksum_coverage(
    struct neat_ctx *ctx,
    struct neat_flow *flow,
    unsigned int send_coverage,
    unsigned int receive_coverage);
```

23.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **send_coverage**: UDP-Lite: The number of bytes covered by the checksum when sending messages. UDP: Ignored.
- **receive_coverage**: UDP-Lite: The lowest number of bytes that must be covered by the checksum on a received message. UDP: See below.

23.3 Return values

- Returns `NEAT_OK` if the checksum coverage was set successfully.
- Returns `NEAT_ERROR_UNABLE` if the checksum coverage cannot be set, either because the value is invalid, or because the protocol does not support it.

23.4 Remarks

Only available for flows using UDP or UDP-Lite.

Checksum verification may be enabled disabled on the receive side for flows using UDP. Specifying a non-zero value for **receive_coverage** will enable it; specifying `0` will disable it.

23.5 Examples

None.

23.6 See also

None.

CHAPTER 24

neat_set_qos

Set the Quality-of-Service class for this flow.

24.1 Syntax

```
neat_error_code neat_set_qos(struct neat_ctx *ctx,
                             struct neat_flow *flow,
                             uint8_t qos);
```

24.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **qos**: The QoS class to use for this flow.

24.3 Return values

- Returns `NEAT_OK` if the QoS class was set successfully.
- Returns `NEAT_ERROR_UNABLE` if NEAT was not able to set the requested QoS class.

24.4 Remarks

None.

24.5 Examples

None.

24.6 See also

None.

CHAPTER 25

neat_get_qos

Get the Quality-of-Service class for this flow.

25.1 Syntax

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

25.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.

25.3 Return values

- Returns the used DiffServ Code Point(DSCP) used signal QoS for this flow.

25.4 Remarks

None.

25.5 Examples

None.

25.6 See also

None.

CHAPTER 26

neat_set_ecn

Set the Explicit Congestion Notification value for this flow.

26.1 Syntax

```
neat_error_code neat_set_ecn(struct neat_ctx *ctx,
                             struct neat_flow *flow,
                             uint8_t ecn);
```

26.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **ecn**: The ECN value to use for this flow.

26.3 Return values

- Returns `NEAT_OK` if the QoS class was set successfully.
- Returns `NEAT_ERROR_UNABLE` if NEAT was not able to set the requested ECN value.

26.4 Remarks

None.

26.5 Examples

None.

26.6 See also

None.

CHAPTER 27

neat_start_event_loop

Starts the internal event loop within NEAT.

27.1 Syntax

```
neat_error_code neat_start_event_loop(struct neat_ctx *ctx, neat_run_mode run_
↪mode);
```

27.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **run_mode**: The mode of which the event loop in NEAT should execute. May be one of either `NEAT_RUN_DEFAULT`, `NEAT_RUN_ONCE`, or `NEAT_RUN_NOWAIT`.

27.3 Return values

- Returns `NEAT_OK` if the NEAT executed with no error.
- Returns an error value if the internal event loop in NEAT was stopped due to an error.

27.4 Remarks

This function does not return when executed with `NEAT_RUN_DEFAULT`.

When executed with `NEAT_RUN_ONCE`, NEAT will poll for I/O, and then block *unless* there are pending callbacks within NEAT that are ready to be processed. These callbacks may be internal.

When executed with `NEAT_RUN_NOWAIT`, NEAT will poll for I/O and execute any pending callbacks. If there are no pending callbacks, it returns after polling.

27.5 Examples

None.

27.6 See also

- *neat_stop_event_loop*
- *neat_get_backend_fd*

CHAPTER 28

neat_stop_event_loop

Stops the internal NEAT event loop.

28.1 Syntax

```
int neat_stop_event_loop(struct neat_ctx *ctx);
```

28.2 Parameters

- **ctx**: Pointer to a NEAT context.

28.3 Return values

None.

28.4 Remarks

Once called, no further events will be processed and no callbacks will be called until `neat_start_event_loop` is called again.

28.5 Examples

None.

28.6 See also

- *neat_start_event_loop*

CHAPTER 29

neat_get_backend_fd

Stops the internal NEAT event loop.

29.1 Syntax

```
int neat_get_backend_fd(struct neat_ctx *ctx);
```

29.2 Parameters

- **ctx**: Pointer to a NEAT context.

29.3 Return values

Returns the file descriptor of the event loop used internally by NEAT. May be polled to check for any new events.

29.4 Remarks

Note that embedding this event loop inside another event loop may not be supported on all systems.

29.5 Examples

None.

29.6 See also

- *neat_start_event_loop*

CHAPTER 30

neat_get_backend_timeout

Return the timeout that should be used when polling the backend file descriptor.

30.1 Syntax

```
int neat_get_backend_timeout(struct neat_ctx *ctx);
```

30.2 Parameters

- **ctx**: Pointer to a NEAT context.

30.3 Return values

Returns the number of milliseconds on which a poll operation may at most be blocked on the backend file descriptor from libuv before the NEAT event loop should be executed again to take care of timer events within NEAT.

30.4 Remarks

The `client_http_run_once` example demonstrates the use of this function.

30.5 Examples

None.

30.6 See also

- *neat_get_backend_fd*

CHAPTER 31

neat_get_event_loop

Return the internal NEAT event loop pointer.

31.1 Syntax

```
uv_loop_t neat_get_event_loop(struct neat_ctx *ctx);
```

31.2 Parameters

- **ctx**: Pointer to a NEAT context.

31.3 Return values

Returns the the event loop used internally by NEAT.

31.4 Examples

None.

31.5 See also

- *neat_start_event_loop*

CHAPTER 32

neat_get_stats

Return statistics from a NEAT context.

32.1 Syntax

```
neat_error_code neat_get_stats(
    struct neat_ctx *ctx,
    char **json_stats);
```

32.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **json_stats**: Pointer to an address where address of the statistics may be written.

32.3 Return values

- Returns `NEAT_OK`.

32.4 Remarks

The statistics is output in JSON format. The caller is responsible for freeing the buffer containing the statistics.

32.5 Examples

None.

32.6 See also

None.

CHAPTER 33

neat_getlpaddrs

Obtains the local or peer addresses of a flow.

33.1 Syntax

```
int neat_getlpaddrs(struct neat_ctx*  ctx,
                    struct neat_flow* flow,
                    struct sockaddr** addrs,
                    const int         local)
```

33.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **flow**: Pointer to a NEAT flow.
- **addrs**: Pointer to variable for storing pointer to addresses to.
- **local**: Set to non-zero value for obtaining local addresses, set to 0 to obtain peer addresses.

33.3 Return values

On success, neat_getlpaddrs() returns the number of addresses (local or remote). In case of having obtained at least one address, a pointer to a newly allocated memory area with the addresses will be stored into addrs. This memory area needs to be freed after usage.

33.4 Examples

```
struct struct sockaddr* addrs;
int n = neat_getlpaddrs(ctx, flow, &addrs, 1);
if(n > 0) {
   struct sockaddr* a = addrs;
```

```
    for(int i = 0; i < n; i++) {
        switch(a->sa_family) {
            case AF_INET:
                printf("Address %d/%d: IPv4\n", i, n);
                a = (struct sockaddr*)((long)a + (long)sizeof(sockaddr_in));
             break;
            case AF_INET6:
                printf("Address %d/%d: IPv6\n", i, n);
                a = (struct sockaddr*)((long)a + (long)sizeof(sockaddr_in6));
            default:
                assert(false);
             break;
        }
    }
    free(addrs);
}
```

CHAPTER 34

neat_log_level

Set the log-level of the NEAT library.

34.1 Syntax

```
void neat_log_level(struct neat_ctx *ctx,
                    uint8_t level)
```

34.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **level**: Log level of the log entry
  - NEAT_LOG_OFF
  - NEAT_LOG_ERROR
  - NEAT_LOG_WARNING
  - NEAT_LOG_ERROR
  - NEAT_LOG_DEBUG

34.3 Return values

None.

34.4 Examples

```
neat_log_level(ctx, NEAT_LOG_ERROR);
```

34.5 See also

- *neat_log_file*

CHAPTER 35

neat_log_file

Sets the name of the log file.

35.1 Syntax

```
uint8_t neat_log_file(struct neat_ctx *ctx,
                      const char* file_name)
```

35.2 Parameters

- **ctx**: Pointer to a NEAT context.
- **file_name**: Name of the NEAT logfile. If set to `NULL`, NEAT writes the log output to `stderr`.

35.3 Return values

- **RETVAL_SUCCESS**: success
- **RETVAL_FAILURE**: failure

35.4 Examples

```
neat_log_file(ctx, "disaster.log");
```

35.5 See also

- *neat_log_level*

CHAPTER 36

Coding Style

The coding style used in NEAT is based on the coding style used in the Linux kernel. There are, however, some differences between the kernel style and the style used in the NEAT project. This document details these differences.

36.1 Strictness

This coding style serves as a guideline. Adherence is not strictly required, but (new) code should still try to follow these guidelines as far as possible to ensure that the code in the NEAT library has a coherent style.

36.2 Disambiguation

If some piece of code does not follow these guidelines, try to match the surrounding code. Do not mix style changes into commits with a different purpose.

36.3 Indentation

4 spaces.

36.4 Line length

There is no strict limit on the length of a line. As a general guideline, try to keep it below 120 characters.

36.5 Placement of braces

Generally the same as the Linux kernel style.

Braces may be used for blocks containing only one statement. Be consistent in adjacent blocks.

Yes:

```
if (this) {
    do_that();
} else {
    something();
}
```

No:

```
if (this) {
    do_that();
} else
    something();
```

36.6 Spaces

As in the Linux kernel style.

Please avoid trailing whitespace.

36.7 Naming

Use names separated by underscores, e.g. `this_is_my_variable`.

Descriptive names are preferred, but short, well-known abbreviations are acceptable.

36.8 Functions

The return type is placed on a separate line preceding the function name:

```
void
my_function(int parameter1, int parameter2)
{

}
```

Align parameters on subsequent lines with the first parameter:

```
void
my_other_function(int parameter1, int parameter2, ...
                  int parameterN, int parameterM)
{

}
```

36.9 Tools

The `.editorconfig` file in the NEAT repository can be used by most editors with the help of a plugin. See www.editorconfig.org.

The `uncrustify-neat.cfg` file can be used by the Uncrustify tool to format source code in accordance with this NEAT style guide. See the Uncrustify documentation for more information.