NEATPy

Release 1.0.0

Michael Gundersen

Nov 18, 2020

API DOCUMENTATION

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

Missing implementation details:

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

- *Selection Properties:*

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

- *Connection Properties:*

Retransmission Threshold Before Excessive Retransmission Notification
Connection Group Transmission Scheduler

- *Events:*

Soft Errors
Excessive retransmissions

- *Security Parameters:*

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

- *Rendezvous Action:*

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

Deviations:

- `Preconnection.start()`:

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

CHAPTER

ONE

API

1.1 Preconnection

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

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

Parameters

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

add_framer(*framer*)

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

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

Return type None

initiate(*timeout=None*)

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

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

Return type Connection

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

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

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

Parameters

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

Return type `Connection`

listen()

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

Return type `Listener`

Returns A listener object.

start()

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

Return type `None`

Returns This function does not return.

1.2 Connection

class ConnectionState(*value*)

An enumeration of the different states for a connection.

CLOSED = 4

CLOSING = 3

ESTABLISHED = 2

ESTABLISHING = 1

class MessageDataObject(*data*, *length*)

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

data: bytearray

The raw bytes of the message

length: int

The message length

class Connection

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

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

HANDLE_STATE_CLOSED: Callable[[Connection], None]
Handler for when the connection transitions to closed state

HANDLE_STATE_CONNECTION_ERROR: Callable[[Connection], None]
Handler for when the connection gets experiences a connection error

HANDLE_STATE_READY: Callable[[Connection], None]
Handler for when the connection transitions to ready state

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

Return type None

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

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

Return type None

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

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

Return type None

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

Return type None

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

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

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

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

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

Return type None

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

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

Parameters

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

Return type None

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

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

Parameters

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

Return type None

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

Parameters

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

Return type None

1.3 Transport Properties

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

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

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

Parameters

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

Return type None

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

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

Return type None

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

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

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

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

Return type None

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

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

Return type None

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

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

Return type None

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

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

Return type None

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

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

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

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

1.4 Selection Properties

class PreferenceLevel(*value*)
An enumeration. Used when specifying a preference for selection_properties.

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

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

AVOID = -1

IGNORE = 0

PREFER = 1

PROHIBIT = -2

REQUIRE = 2

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

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

DIRECTION = 'direction'
Default communication_directions

INTERFACE = 'interface'
Default preference_level

LOCAL_ADDRESS_PREFERENCE = 'local-address-preference'
Default preference_level

MULTIPATH = 'multipath'
Default preference_level.PREFER

MULTISTREAMING = 'multistreaming'
Default preference_level.PREFER,

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

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

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

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

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

PVD = 'pvd'
Default preference_level

RELIABILITY = 'reliability'
Default preference_level.REQUIRE,

RETRANSMIT_NOTIFY = 'retransmit-notify'
Default preference_level.IGNORE

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

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

1.5 Connection Properties

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

PRIORITY = 'conn-prio'
Default value is 100

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

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

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

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

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

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

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

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

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

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

1.6 Message Properties

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

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

EARLY_DATA = 'early-data'
Read only property

ECN = 'ecn'
Read only property

FINAL = 'final'
Default value is False

IDEMPOTENT = 'idempotent'
Default value is False

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

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

ORDERED = 'msg-ordered'
Default value is True

PRIORITY = 'msg-prio'
Default value is 100

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

RELIABLE_DATA_TRANSFER = 'msg-reliable'
Default value is True

SINGULAR_TRANSMISSION = 'singular-transmission'
Default value is False

1.7 Endpoints

class LocalEndpoint

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

with_address(*address*)

This function sets the address desired to use with the local endpoint.

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

Return type None

with_interface(*interface*)

This function sets the interface desired to with the local endpoint.

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

Return type None

with_port(*port_number*)

This function sets the port desired to use with the local endpoint.

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

Return type None

class RemoteEndpoint

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

with_address(*address*)

This function sets the address desired to use with the local endpoint.

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

Return type None

with_hostname(*hostname*)

This function sets the hostname for the remote endpoint.

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

Return type None

with_interface(*interface*)

This function sets the interface desired to with the local endpoint.

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

Return type None

with_port(*port_number*)

This function sets the port desired to use with the local endpoint.

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

Return type None

1.8 Framer

class Framer

abstract handle_received_data(*connection*)
Upon receiving this event, the framer implementation can inspect the inbound data. The data is parsed from a particular cursor representing the unprocessed data. The application requests a specific amount of data it needs to have available in order to parse. If the data is not available, the parse fails.

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

abstract new_sent_message(*connection*, *message_data*, *message_context*, *sent_handler*, *is_end_of_message*)
Upon receiving this event, a framer implementation is responsible for performing any necessary transformations and sending the resulting data back to the Message Framer, which will in turn send it to the next protocol.

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

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

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

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

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

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

CHAPTER

TWO

CLIENT-SERVER EXAMPLE

Let us create a simple client and server with NEATPy!

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

Let us start with the server!

2.1 Server

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
import neatpy

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

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

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

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

    new_listener.HANDLE_CONNECTION_RECEIVED = new_connection_received
    new_preconnection.start()

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

2.2 Client

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

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

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

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

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

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

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

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

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

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

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

Our client in full looks like the following:

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

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

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

    transport_properties = TransportProperties(TransportPropertyProfiles.RELIABLE_INORDER_STREAM)

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

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

PYTHON MODULE INDEX

INDEX

N

O

P

R

S

T

U

W

Z