ncplib Documentation

Release 6.1.0

Dave Hall

Jun 11, 2022

Contents

NCP library for Python 3, developed by CRFS.

CHAPTER 1

Features

- *NCP client*.
- *NCP server*.
- Asynchronous connections via `asyncio`.

CHAPTER 2

Resources

- Documentation is on Read the Docs.
- Examples, issue tracking and source code are on GitHub.

CHAPTER 3

Usage

3.1 Installation

3.1.1 Requirements

ncplib supports Python 3.7 and above. It has no other dependencies.

3.1.2 Installing

It's recommended to install *ncplib* in a virtual environment using venv.

Install *ncplib* using pip.

```
pip install ncplib
```

3.1.3 Upgrading

Upgrade *ncplib* using pip:

```
pip install --upgrade ncplib
```

Important: Check the *Changelog* before upgrading.

3.2 NCP client

ncplib allows you to connect to a *NCP server* and issue commands.

3.2.1 Overview

Connecting to a NCP server

Connect to a *NCP server* using `connect()`. The returned `Connection` will automatically close when the connection block exits.

```
import ncplib

async with await ncplib.connect("127.0.0.1", 9999) as connection:
    pass  # Your client code here.

# Connection is automatically closed here.
```

Sending a packet

Send a *NCP packet* containing a single *NCP field* using `Connection.send()`:

```
response = connection.send("DSPC", "TIME", SAMP=1024, FCTR=1200)
```

Receiving replies to a packet

The return value of `Connection.send()` is a `Response`. Receive a single *NCP field* reply using `Response.recv()`:

```
field = await response.recv()
```

Alternatively, use the `Response` as an *async iterator* to loop over multiple *NCP field* replies:

```
async for field in response:
    pass
```

Important: The *async for loop* will only terminate when the underlying connection closes.

Accessing field data

The return value of `Response.recv()` is a `Field`, representing a *NCP field*. Access contained *NCP parameters* using item access:

```
print(field["TSDC"])
```

Advanced usage

- *NCP connection documentation*.

3.2.2 API reference

ncplib.**connect**(*host: str*, *port: Optional[int] = None*, ***, *remote_hostname: Optional[str] = None*, *hostname: Optional[str] = None*, *connection_username: Optional[str] = None*, *connection_domain: str = ''*, *timeout: int = 60*, *auto_erro: bool = True*, *auto_warn: bool = True*, *auto_ackn: bool = True*, *ssl: bool | ssl.SSLContext = False*, *username: str = ''*, *password: str = ''*) → Connection

Connects to a *NCP server*.

Parameters

- **host** (*str*) – The hostname of the *NCP server*. This can be an IP address or domain name.
- **port** (*int*) – The port number of the *NCP server*.
- **remote_hostname** (*str*) – The identifying hostname for the remote end of the connection. If omitted, this will be the host:port of the NCP server.
- **hostname** (*str*) – The identifying hostname in the client connection. Defaults to the system hostname.
- **connection_username** (*str*) – The identifying username in the client connection. Defaults to the login name of the system user.
- **connection_domain** (*str*) – The identifying domain in the client connection.
- **timeout** (*int*) – The network timeout (in seconds). Applies to: connecting, receiving a packet, closing connection.
- **auto_erro** (*bool*) – Automatically raise a *CommandError* on receiving an ERRO *NCP parameter*.
- **auto_warn** (*bool*) – Automatically issue a *CommandWarning* on receiving a WARN *NCP parameter*.
- **auto_ackn** (*bool*) – Automatically ignore *NCP fields* containing an ACKN *NCP parameter*.
- **ssl** (*bool*) – Connect to the Node using an encrypted (TLS) connection. Requires TLS support on the Node.
- **username** (*str*) – Authenticate with the Node using the given username. Requires authentication support on the Node.
- **password** (*str*) – Authenticate with the Node using the given password. Requires authentication support on the Node.

Raises ***ncplib.NCPError*** – if the NCP connection failed.

Returns The client *Connection*.

Return type *Connection*

3.3 NCP server

ncplib allows you to create a NCP server and respond to incoming *NCP client* connections.

3.3.1 Overview

Defining a connection handler

A connection handler is a coroutine that starts whenever a new *NCP client* connects to the server. The provided `Connection` allows you to receive incoming NCP commands as `Field` instances.

```
async def client_connected(connection):
    pass
```

When the connection handler exits, the `Connection` will automatically close.

Listening for an incoming packet

When writing a *NCP server*, you most likely want to wait for the connected client to execute a command. Within your `client_connected` function, Listen for an incomining *NCP field* using `Connection.recv()`.

```
field = await connection.recv()
```

Alternatively, use the `Connection` as an *async iterator* to loop over multiple *NCP field* replies:

```
async for field in connection:
    pass
```

Important: The *async for loop* will only terminate when the underlying connection closes.

Accessing field data

The return value of `Connection.recv()` is a `Field`, representing a *NCP field*.

Access information about the *NCP field* and enclosing *NCP packet*:

```
print(field.packet_type)
print(field.name)
```

Access contained *NCP parameters* using item access:

```
print(field["FCTR"])
```

Replying to the incoming field

Send a reply to an incoming `Field` using `Field.send()`.

```
field.send(ACKN=1)
```

Putting it all together

A simple `client_connected` callback might like this:

```
async def client_connected(connection):
    async for field in connection:
        if field.packet_type == "DSPC" and field.name == "TIME":
            field.send(ACNK=1)
            # Do some more command processing here.
        else:
            field.send(ERRO="Unknown command", ERRC=400)
            break
```

Start the server

Start a new NCP server.

```
loop = asyncio.get_event_loop()
server = loop.run_until_complete(_start_server(client_connected))
try:
    loop.run_forever()
finally:
    server.close()
    loop.run_until_complete(server.wait_closed())
```

Advanced usage

- *NCP connection documentation*.

3.3.2 API reference

ncplib.**start_server**(*client_connected: Callable[[ncplib.connection.Connection], Awaitable[None]], host: str = '0.0.0.0', port: Optional[int] = None, *, timeout: int = 60, start_serving: bool = True, ssl: Optional[ssl.SSLContext] = None, authenticate: Optional[Callable[[str, str], bool]] = None*) → asyncio.base_events.Server

Creates and returns a new `Server` on the given host and port.

Parameters

- **client_connected** – A coroutine function taking a single *Connection* argument representing the client connection. When the connection handler exits, the *Connection* will automatically close. If the client closes the connection, the connection handler will exit.
- **host** (*str*) – The host to bind the server to.
- **port** (*int*) – The port to bind the server to.
- **timeout** (*int*) – The network timeout (in seconds). Applies to: creating server, receiving a packet, closing connection, closing server.
- **start_serving** (*bool*) – Causes the created server to start accepting connections immediately.
- **ssl** (*ssl.SSLContext*) – Start the server using an encrypted (TLS) connection.
- **authenticate** – A callable taking a username and password argument, returning True if the authentication is successful, and false if not. When present, authentication is mandatory.

Returns The created `Server`.

Return type Server

3.4 NCP connection

NCP connections are used by the *NCP client* and *NCP server* to represent each side of a connection.

3.4.1 Overview

Getting started

- *NCP client documentation*.
- *NCP server documentation*.

Spawning tasks

Spawn a concurrent task to handle long-running commands:

```
import asyncio

loop = asyncio.get_event_loop()

async def handle_dspc_time(field):
    field.send(ACKN=1)
    await asyncio.sleep(10)  # Simulate a blocking task.
    field.send(TSDC=0, TIMM=1)

for field in connection:
    if field.packet_type == "DSPC" and field.name == "TIME":
        # Spawn a concurrent task to avoid blocking the accept loop.
        loop.create_task(handle_dspc_time(field))
    # Handle other field types here.
```

3.4.2 API reference

Important: Do not instantiate these classes directly. Use `connect()` to create a *NCP client* connection. Use `start_server()` to create a *NCP server*.

class ncplib.**Connection**(*reader: asyncio.streams.StreamReader, writer: asyncio.streams.StreamWriter, predicate: Callable[[ncplib.connection.Field], bool], *, logger: logging.Logger, remote_hostname: str, timeout: int*)

A connection between a *NCP client* and a *NCP server*.

Connections can be used as *async iterators* to loop over each incoming `Field`:

```
async for field in connection:
    pass
```

Important: The *async for loop* will only terminate when the underlying connection closes.

Connections can also be used as *async context managers* to automatically close the connection:

```
async with connection:
    pass

# Connection is automatically closed.
```

logger
The `logging.Logger` used by this connection. Log messages will be prefixed with the host and port of the connection.

remote_hostname
The identifying hostname for the remote end of the connection.

close() → None
Closes the connection.

Hint: If you use the connection as an *async context manager*, there's no need to call *`Connection.close()`* manually.

---

is_closing() → bool
Returns True if the connection is closing.

A closing connection should not be written to.

recv() → ncplib.connection.Field
Waits for the next *`Field`* received by the connection.

Raises ***ncplib.NCPError*** – if a field could not be retrieved from the connection.

Returns The next *`Field`* received.

Return type *Field*

recv_field(*packet_type: str*, *field_name: str*) → ncplib.connection.Field
Waits for the next matching *`Field`* received by the connection.

Parameters

- **packet_type** (*`str`*) – The packet type, must be a valid *identifier*.
- **field_name** (*`str`*) – The field name, must be a valid *identifier*.

Raises ***ncplib.NCPError*** – if a field could not be retrieved from the connection.

Returns The next *`Field`* received.

Return type *Field*

send(*packet_type: str*, *field_name: str*, ***params*) → ncplib.connection.Response
Sends a *NCP packet* containing a single *NCP field*.

Parameters

- **packet_type** (*`str`*) – The packet type, must be a valid *identifier*.
- **field_name** (*`str`*) – The field name, must be a valid *identifier*.
- ****params** – Keyword arguments, one per *NCP parameter*. Each parameter name should be a valid *identifier*, and each parameter value should be one of the supported *value types*.

Returns A *`Response`* providing access to any *`Field`* instances received in reply to the sent packet.

Return type *Response*

Raises

- **ValueError** – if any of the packet, field or parameter names were not a valid *identifier*, or any of the parameter values were invalid.
- **TypeError** – if any of the parameter values were not one of the supported *value types*.

send_packet(*packet_type: str*, ***fields*) → ncplib.connection.Response
Sends a *NCP packet* containing multiple *NCP fields*.

Hint: Prefer *send()* unless you need to send multiple fields in a single packet.

Parameters

- **packet_type** (*str*) – The packet type, must be a valid *identifier*.
- ****fields** – Keyword arguments, one per field. Each field name should be a valid *identifier*, and the field value should be a dict of parameter names mapped to parameter values. Each parameter name should be a valid *identifier*, and each parameter value should be one of the supported *value types*.

Returns A *Response* providing access to any *Field* instances received in reply to the sent packet.

Return type *Response*

Raises

- **ValueError** – if any of the packet, field or parameter names were not a valid *identifier*, or any of the parameter values were invalid.
- **TypeError** – if any of the parameter values were not one of the supported *value types*.

transport
The asyncio.WriteTransport used by this connection.

wait_closed() → None
Waits for the connection to finish closing.

Hint: If you use the connection as an *async context manager*, there's no need to call *Connection.wait_closed()* manually.

class ncplib.**Response**(*connection: ncplib.connection.Connection, packet_type: str, expected_fields: Set[Tuple[str, int]]*)
A response to a *NCP packet*, returned by *Connection.send()*, *Connection.send_packet()* and *Field.send()*.

Provides access to any *Field* received in reply to the sent packet.

Responses can be used as *async iterators* to loop over each incoming *Field*:

```
async for field in response:
    pass
```

Important: The *async for loop* will only terminate when the underlying connection closes.

recv() → ncplib.connection.Field
Waits for the next *Field* received in reply to the sent *NCP packet*.

Raises ***ncplib.NCPError*** – if a field could not be retrieved from the connection.

Returns The next *Field* received.

Return type *Field*

recv_field(*field_name: str*) → ncplib.connection.Field
Waits for the next matching *Field* received in reply to the sent *NCP packet*.

Hint: Prefer *recv()* unless the sent packet contained multiple fields.

Parameters **field_name** (*str*) – The field name, must be a valid *identifier*.

Raises ***ncplib.NCPError*** – if a field could not be retrieved from the connection.

Returns The next *Field* received.

Return type *Field*

class ncplib.**Field**(*connection: ncplib.connection.Connection, packet_type: str, packet_id: int, packet_timestamp: datetime.datetime, name: str, id: int, params: Iterable[Tuple[str, Union[bytes, bytearray, str, int, float, ncplib.values.u32, ncplib.values.i64, ncplib.values.u64, ncplib.values.f64, bool, array.array]]]*)
A *NCP field* received by a *Connection*.

Access *NCP parameter* values using item access:

```
print(field["PDAT"])
```

connection
The *Connection* that created this field.

packet_type
The type of *NCP packet* that contained this field. This will be a valid *identifier*.

packet_id
The ID of the of *NCP packet* that contained this field.

packet_timestamp
A timezone-aware `datetime.datetime` describing when the containing packet was sent.

name
The name of the *NCP field*. This will be a valid *identifier*.

id
The unique `int` ID of this field.

send(***params*) → ncplib.connection.Response
Sends a *NCP packet* containing a single field in reply to this field.

Parameters *****params*** – Keyword arguments, one per *NCP parameter*. Each parameter name should be a valid *identifier*, and each parameter value should be one of the supported *value types*.

Returns A *Response* providing access to any *Field* instances received in reply to the sent packet.

Return type *Response*

Raises

- **ValueError** – if any of the packet, field or parameter names were not a valid *identifier*, or any of the parameter values were invalid.
- **TypeError** – if any of the parameter values were not one of the supported *value types*.

3.5 Errors and warnings

NCP errors and warnings.

3.5.1 API reference

exception ncplib.**NCPError**
Base class for all exceptions thrown by *ncplib*.

exception ncplib.**NetworkError**
Raised when an NCP *Connection* cannot connect, or disconnects unexpectedly.

exception ncplib.**AuthenticationError**
Raised when an NCP *Connection* cannot authenticate.

exception ncplib.**NetworkTimeoutError**
Raised when an NCP *Connection* times out while performing network activity.

exception ncplib.**ConnectionClosed**
Raised when an NCP *Connection* is closed gracefully.

exception ncplib.**CommandError**(*field: Field*, *detail: str*, *code: int*)
Raised by the *NCP client* when the *NCP server* sends a *NCP field* containing an ERRO parameter.

Can be disabled by setting auto_erro to False in *ncplib.connect()*.

field
The *ncplib.Field* that triggered the error.

detail
The human-readable str message from the server.

code
The int code from the server,

exception ncplib.**DecodeError**
Raised when a non-recoverable error was encountered in a *NCP packet*.

exception ncplib.**CommandWarning**(*field: Field*, *detail: str*, *code: int*)
Issued by the *NCP client* when the *NCP server* sends a *NCP field* containing a WARN parameter.

Can be disabled by setting auto_warn to False in *ncplib.connect()*.

field
The *ncplib.Field* that triggered the error.

detail
The human-readable str message from the server.

code
The int code from the server,

exception ncplib.**DecodeWarning**
Issued when a recoverable error was encountered in a *NCP packet*.

3.6 Value types

3.6.1 Overview

NCP data types are mapped onto python types as follows:

NCP type	Python type
i32	`int`
u32	*`ncplib.u32`*
i64	*`ncplib.i64`*
u64	*`ncplib.u64`*
f32	`float`
f64	*`ncplib.f64`*
string	`str`
raw	`bytes`
data i8	`array.array(typecode="b")`
data i16	`array.array(typecode="h")`
data i32	`array.array(typecode="i")`
data u8	`array.array(typecode="B")`
data u16	`array.array(typecode="H")`
data u32	`array.array(typecode="I")`
data u64	`array.array(typecode="L")`
data i64	`array.array(typecode="l")`
data f32	`array.array(typecode="f")`
data f64	`array.array(typecode="d")`

3.6.2 API reference

class ncplib.**u32**
A *u32* value.

Wrap any `int` values to be encoded as `u32` in *u32*.

class ncplib.**i64**
An *i64* value.

Wrap any `int` values to be encoded as `i64` in *i64*.

class ncplib.**u64**
A *u64* value.

Wrap any `int` values to be encoded as `u64` in *u64*.

class ncplib.**f64**
A *f64* value.

Wrap any `float` values to be encoded as `f64` in *f64*.

CHAPTER 4

More information

4.1 Contributing

Bug reports, bug fixes, and new features are always welcome. Please raise issues on GitHub, and submit pull requests for any new code.

4.1.1 Testing

It's recommended to test *ncplib* in a virtual environment using `venv`.

Run the test suite:

```
pip install -e .
python -m unittest discover tests
```

4.1.2 Contributors

ncplib was developed by CRFS and other contributors.

4.2 Glossary

identifier A `str` of ascii uppercase letters and numbers, at most 4 characters long, e.g. `"DSPC"`.

NCP Node Communication Protocol, a binary communication and control protocol, developed by CRFS.

NCP field Each *NCP packet* contains zero or more fields. A field consists of a field *name*, which must be a valid *identifier*, and zero or more *NCP parameters*.

ncplib represents each field in an incoming *NCP packet* as a *ncplib.Field* instance.

NCP packet The basic unit of *NCP* communication. A packet consists of a packet *type*, which must be a valid *identifier*, and zero or more *NCP fields*.

NCP parameter Each *NCP field* contains zero or more parameters. A parameter consists of a parameter *name*, which must be a valid *identifier*, and a *value*, which must be one of the supported *value types*.

`ncplib` represents each parameter as a name/value mapping on a `ncplib.Field` instance.

4.3 Changelog

4.3.1 6.1.0 - 11/06/2022

- Added `connection_username` and `connection_domain` arguments to `connect()`.

4.3.2 6.0.1 - 23/12/2021

- Added explicit support for `async_timeout` 4.0.

4.3.3 6.0.0 - 23/12/2021

- Added support for NCP encrypted (TLS) connections via the `ssl` argument for `connect()` and `start_server()`.
- Added support for NCP authentication via the `username` and `password` arguments for `connect()`, and the `authenticate` argument for `start_server()`.
- **Breaking:** `start_server()` now returns a `asyncio.base_events.Server`, and the `ncplib.Server` class has been removed.

4.3.4 5.0.0 - 18/02/2021

- Added support for NCP connection timeout negotation, improving reliability and cleanup of NCP connections when supported by the remote.
- Added support for NCP data types `i64`, `u64`, `f32`, `f64`, `data u64`, `data i64`, `data f32` and `data f64`.
- `Response.recv()` no longer requires the ID of the of *NCP packet* in replies.
- **Breaking:** `auto_link` and `auto_auth` arguments for `connect()` and `start_server()` removed.
- **Breaking:** `timeout` argument for `connect()` and `start_server()` must be an integer, and can no longer be None.
- **Breaking:** Removed `timeout` attribute from `Connection`.

4.3.5 4.1.1 - 14/09/2020

- Optimized `auto_link` background task.

4.3.6 4.1.0 - 07/07/2020

- Added `Field.packet_id` attribute.
- *`Field.send()`* now includes the ID of the of *NCP packet* that contained the field.
- *`Response.recv()`* now requires the ID of the of *NCP packet* in replies.

4.3.7 4.0.0 - 20/05/2020

- **Breaking:** Renamed `ConnectionError` to *`NetworkError`* to avoid conflicts with stdlib.
- Added *timeout* parameter to *`connect()`*, *`start_server()`* and *`Connection`*. This is the network timeout (in seconds). If *None*, no timeout is used, which can lead to deadlocks. The default timeout is 15 seconds. A *`NetworkTimeoutError`* error will be raised if a timeout is exceeded.

4.3.8 3.0.0 - 24/10/2019

This release requires a minimum Python version of 3.7.

- **Breaking:** Python 3.7 is now the minimum supported Python version.
- **Breaking:** Removed `app` framework.
- **Breaking:** Removed `run_client` and `run_app`.
- Added *`Connection.wait_closed()`* to ensure that the connection is fully closed (needed since Python 3.7).
- Added full PEP 484 type hints, allowing tools like *mypy* to be used to statically-verify *ncplib* programs.

4.3.9 2.3.3 - 27/03/2017

- Only applying wait_for compatibility shim to Python 3.4.2 and below.

4.3.10 2.3.2 - 15/03/2017

- Forcing cancellation of timed out connection in `run_client` in Python 3.4.2.
- Added examples.

4.3.11 2.3.1 - 02/03/2017

- Using `remote_hostname` in connect errors messages generated by `run_client`.
- Fixed issues with mixing coroutines and async defs.
- Fixed issues with logging connection errors in `run_client`.

4.3.12 2.3.0 - 02/03/2017

- Added *Field.connection*.
- Added app.
- Added :class‘NCPError‘, ConnectionError and :class‘ConnectionClosed‘ exceptions.
- Added run_client.
- *connect()*, *Connection.recv()*, *Connection.recv_field()*, *Response.recv()* and *Response.recv_field()* no longer raise EOFError or OSError, but a subclass of *NCPError*.
- Micro-optimizations, roughly doubling the performance of encode/decode.
- Connection open and close log messages promoted from DEBUG to INFO level.

4.3.13 2.2.1 - 27/02/2017

- Fixed bug with Node authentication due to premature sending of LINK packets.
- Fixed edge-case bug in connection closing.

4.3.14 2.2.0 - 27/02/2017

- Added Python 3.4 support.
- Added *Connection.is_closing()*.
- Added *Connection.remote_hostname*.
- Added auto_link parameter to *connect()*, *start_server()* and run_app.
- Added remote_hostname parameter to *connect()*.
- Connection open and close log messages demoted from INFO to DEBUG level.

4.3.15 2.1.0 - 04/11/2016

- Client hostname used in *connect()* defaults to system hostname, instead of "python3-ncplib".
- Added hostname parameter to *connect()*, to override default client hostname.
- Removed multiplexing support for multiple *Response* over a single connection. This must now be implemented in application code.
- *Connection* logger no longer formats the host and port in log messages. This must now be done using the standard Python logging.Formatter.

4.3.16 2.0.14 - 04/11/2016

- Added support for parsing known embedded footer bug from Axis nodes.
- Fixed pending deprecation warning for legacy __aiter__ protocol.

4.3.17 2.0.13 - 21/10/2016

- Using transport.is_closing() to detect lost connection, making ncplib compatible with uvloop.

4.3.18 2.0.12 - 21/10/2016

- *Connection.recv_field()* and *Response.recv_field()* now raise an exception on network error to match the behavior of *Connection.recv()* and *Response.recv()*. Previously they returned `None` on network error, an undocumented and undesired behavior.

4.3.19 2.0.11 - 14/10/2016

- Deprecated `wait_closed()` on *Connection*. It's now a no-op, and *Connection.close()* is sufficient to close the connection.

4.3.20 2.0.10 - 14/10/2016

- Fixed IPv6 handling in NCP server.

4.3.21 2.0.9 - 13/10/2016

- Handling more classes of shutdown errors.

4.3.22 2.0.8 - 13/10/2016

- Suppressing connection errors in NCP server.

4.3.23 2.0.7 - 13/10/2016

- Handling more classes of shutdown errors.

4.3.24 2.0.6 - 13/10/2016

- Handling more classes of client connection error gracefully.
- Handling shutdown of broken connections gracefully.

4.3.25 2.0.5 - 11/10/2016

- Gracefully closing client connections on authentication error.

4.3.26 2.0.4 - 05/09/2016

- Not validating packet format in incoming packets.

4.3.27 2.0.3 - 02/09/2016

- Not logging client errors and warnings, since raised exceptions/warnings will do this automatically.

4.3.28 2.0.2 - 01/09/2016

- Stripping trailing spaces from field names on decode, in addition to null bytes.

4.3.29 2.0.1 - 19/07/2016

- Added `run_app` function to *NCP server*.

4.3.30 2.0.0 - 17/03/2016

This release requires a minimum Python version of 3.5. This allows *ncplib* to take advantage of new native support for coroutines in Python 3.5. It also provides a new *start_server()* function for creating a *NCP server*.

A number of interfaces have been updated or removed in order to take better advantage of Python 3.5 async features, and to unify the interface between *NCP client* and *NCP server* connections. Please read the detailed release notes below for more information.

- *NCP server* support.
- *Connection* can be used as an *async context manager*.
- *Connection.send()* has a cleaner API, allowing params to be specified as keyword arguments.
- *Connection.send()* and *Connection.send_packet()* return a *Response* that can be used to access replies to the original messages.
- *Connection.recv()*, *Connection.recv_field()*, *Response.recv()* and *Response.recv_field()* return a *Field* instance, representing a *NCP field*.
- *Connection* and *Response* can be used as an *async iterator* of *Field*.
- *Field.send()* allows direct replies to be sent to the incoming *NCP field*.
- **Breaking:** Python 3.5 is now the minimum supported Python version.
- **Breaking:** *Connection.send()* API has changed to be single-field. Use *Connection.send_packet()* to send a multi-field *NCP packet*.
- **Breaking:** `Connection.execute()` has been removed. Use `Connection.send().recv()` instead.

4.3.31 1.0.1 - 21/12/2015

- Automated build and release of package to private Anaconda Cloud channel.

4.3.32 1.0.0 - 07/12/2015

- First production release.

Python Module Index

n

Index