pynsq Documentation

Release 0.7.0

Matt Reiferson and Jehiah Czebotar

December 07, 2016

Contents

The official Python client library for NSQ

It provides high-level *nsq.Reader* and *nsq.Writer* classes for building consumers and producers and two low-level modules for both sync and async communication over the NSQ Protocol (if you wanted to write your own high-level functionality).

The async module is built on top of the Tornado IOLoop and as such requires `tornado` to be installed.

Contents:

CHAPTER 1

`Reader` – high-level consumer

class `nsq.`**`Reader`**(*topic*, *channel*, *message_handler=None*, *name=None*, *nsqd_tcp_addresses=None*, *lookupd_http_addresses=None*, *max_tries=5*, *max_in_flight=1*, *lookupd_poll_interval=60*, *low_rdy_idle_timeout=10*, *max_backoff_duration=128*, *lookupd_poll_jitter=0.3*, *lookupd_connect_timeout=1*, *lookupd_request_timeout=2*, ***kwargs*)

Reader provides high-level functionality for building robust NSQ consumers in Python on top of the async module.

Reader receives messages over the specified `topic/channel` and calls `message_handler` for each message (up to `max_tries`).

Multiple readers can be instantiated in a single process (to consume from multiple topics/channels at once).

Supports various hooks to modify behavior when heartbeats are received, to temporarily disable the reader, and pre-process/validate messages.

When supplied a list of `nsqlookupd` addresses, it will periodically poll those addresses to discover new producers of the specified `topic`.

It maintains a sufficient RDY count based on the # of producers and your configured `max_in_flight`.

Handlers should be defined as shown in the examples below. The handler receives a *`nsq.Message`* object that has instance methods *`nsq.Message.finish()`*, *`nsq.Message.requeue()`*, and *`nsq.Message.touch()`* to respond to `nsqd`.

When messages are not responded to explicitly, it is responsible for sending `FIN` or `REQ` commands based on return value of `message_handler`. When re-queueing, it will backoff from processing additional messages for an increasing delay (calculated exponentially based on consecutive failures up to `max_backoff_duration`).

Synchronous example:

```
import nsq

def handler(message):
    print message
    return True

r = nsq.Reader(message_handler=handler,
        lookupd_http_addresses=['http://127.0.0.1:4161'],
        topic='nsq_reader', channel='asdf', lookupd_poll_interval=15)
nsq.run()
```

Asynchronous example:

```
import nsq

buf = []

def process_message(message):
    global buf
    message.enable_async()
    # cache the message for later processing
    buf.append(message)
    if len(buf) >= 3:
        for msg in buf:
            print msg
            msg.finish()
        buf = []
    else:
        print 'deferring processing'

r = nsq.Reader(message_handler=process_message,
        lookupd_http_addresses=['http://127.0.0.1:4161'],
        topic='nsq_reader', channel='async', max_in_flight=9)
nsq.run()
```

Parameters

- **`message_handler`** – the callable that will be executed for each message received
- **`topic`** – specifies the desired NSQ topic
- **`channel`** – specifies the desired NSQ channel
- **`name`** – a string that is used for logging messages (defaults to 'topic:channel')
- **`nsqd_tcp_addresses`** – a sequence of string addresses of the nsqd instances this reader should connect to
- **`lookupd_http_addresses`** – a sequence of string addresses of the nsqlookupd instances this reader should query for producers of the specified topic
- **`max_tries`** – the maximum number of attempts the reader will make to process a message after which messages will be automatically discarded
- **`max_in_flight`** – the maximum number of messages this reader will pipeline for processing. this value will be divided evenly amongst the configured/discovered nsqd producers
- **`lookupd_poll_interval`** – the amount of time in seconds between querying all of the supplied nsqlookupd instances. a random amount of time based on thie value will be initially introduced in order to add jitter when multiple readers are running
- **`lookupd_poll_jitter`** – The maximum fractional amount of jitter to add to the lookupd pool loop. This helps evenly distribute requests even if multiple consumers restart at the same time.
- **`lookupd_connect_timeout`** – the amount of time in seconds to wait for a connection to `nsqlookupd` to be established
- **`lookupd_request_timeout`** – the amount of time in seconds to wait for a request to `nsqlookupd` to complete.
- **`low_rdy_idle_timeout`** – the amount of time in seconds to wait for a message from a producer when in a state where RDY counts are re-distributed (ie. max_in_flight < num_producers)

- **max_backoff_duration** – the maximum time we will allow a backoff state to last in seconds
- ****kwargs** – passed to *nsq.AsyncConn* initialization

close()
Closes all connections stops all periodic callbacks

connect_to_nsqd(*host*, *port*)
Adds a connection to `nsqd` at the specified address.

Parameters

- **host** – the address to connect to
- **port** – the port to connect to

classmethod disabled()
Called as part of RDY handling to identify whether this Reader has been disabled

This is useful to subclass and override to examine a file on disk or a key in cache to identify if this reader should pause execution (during a deploy, etc.).

Note: deprecated. Use set_max_in_flight(0)

giving_up(*message*)
Called when a message has been received where `msg.attempts > max_tries`

This is useful to subclass and override to perform a task (such as writing to disk, etc.)

Parameters **message** – the *nsq.Message* received

heartbeat(*conn*)
Called whenever a heartbeat has been received

This is useful to subclass and override to perform an action based on liveness (for monitoring, etc.)

Parameters **conn** – the *nsq.AsyncConn* over which the heartbeat was received

is_starved()
Used to identify when buffered messages should be processed and responded to.

When max_in_flight > 1 and you're batching messages together to perform work is isn't possible to just compare the len of your list of buffered messages against your configured max_in_flight (because max_in_flight may not be evenly divisible by the number of producers you're connected to, ie. you might never get that many messages... it's a *max*).

Example:

```
def message_handler(self, nsq_msg, reader):
    # buffer messages
    if reader.is_starved():
        # perform work

reader = nsq.Reader(...)
reader.set_message_handler(functools.partial(message_handler, reader=reader))
nsq.run()
```

process_message(*message*)
Called when a message is received in order to execute the configured `message_handler`

This is useful to subclass and override if you want to change how your message handlers are called.

Parameters **message** – the *nsq.Message* received

`query_lookupd``()`

Trigger a query of the configured `nsq_lookupd_http_addresses`.

`set_max_in_flight``(`*max_in_flight*`)`

dynamically adjust the reader max_in_flight count. Set to 0 to immediately disable a Reader

`set_message_handler``(`*message_handler*`)`

Assigns the callback method to be executed for each message received

Parameters **`message_handler`** – a callable that takes a single argument

`nsq.`**`run`**`()`

Starts any instantiated *`nsq.Reader`* or *`nsq.Writer`*

CHAPTER 2

Writer – high-level producer

class nsq.**Writer**(*nsqd_tcp_addresses*, *reconnect_interval=15.0*, *name=None*, ***kwargs*)

A high-level producer class built on top of the Tornado IOLoop supporting async publishing (PUB & MPUB & DPUB) of messages to nsqd over the TCP protocol.

Example publishing a message repeatedly using a Tornado IOLoop periodic callback:

```
import nsq
import tornado.ioloop
import time

def pub_message():
    writer.pub('test', time.strftime('%H:%M:%S'), finish_pub)

def finish_pub(conn, data):
    print(data)

writer = nsq.Writer(['127.0.0.1:4150'])
tornado.ioloop.PeriodicCallback(pub_message, 1000).start()
nsq.run()
```

Example publshing a message from a Tornado HTTP request handler:

```
import functools
import tornado.httpserver
import tornado.ioloop
import tornado.options
import tornado.web
from nsq import Writer, Error
from tornado.options import define, options

class MainHandler(tornado.web.RequestHandler):
    @property
    def nsq(self):
        return self.application.nsq

    def get(self):
        topic = 'log'
        msg = 'Hello world'
        msg_cn = 'Hello '

        self.nsq.pub(topic, msg) # pub
        self.nsq.mpub(topic, [msg, msg_cn]) # mpub
        self.nsq.dpub(topic, 60, msg) # dpub
```

```
            # customize callback
            callback = functools.partial(self.finish_pub, topic=topic, msg=msg)
            self.nsq.pub(topic, msg, callback=callback)

            self.write(msg)

        def finish_pub(self, conn, data, topic, msg):
            if isinstance(data, Error):
                # try to re-pub message again if pub failed
                self.nsq.pub(topic, msg)

    class Application(tornado.web.Application):
        def __init__(self, handlers, **settings):
            self.nsq = Writer(['127.0.0.1:4150'])
            super(Application, self).__init__(handlers, **settings)
```

Parameters

- **`nsqd_tcp_addresses`** – a sequence with elements of the form 'address:port' corresponding to the `nsqd` instances this writer should publish to
- **`name`** – a string that is used for logging messages (defaults to first nsqd address)
- **`**kwargs`** – passed to *`nsq.AsyncConn`* initialization

`heartbeat`(*conn*)

Called whenever a heartbeat has been received

This is useful to subclass and override to perform an action based on liveness (for monitoring, etc.)

Parameters **`conn`** – the *`nsq.AsyncConn`* over which the heartbeat was received

`nsq.`**`run`**()

Starts any instantiated *`nsq.Reader`* or *`nsq.Writer`*

`AsyncConn` – a connection to nsqd

class `nsq.`**`AsyncConn`**(*host*, *port*, *timeout=1.0*, *heartbeat_interval=30*, *requeue_delay=90*, *tls_v1=False*, *tls_options=None*, *snappy=False*, *deflate=False*, *deflate_level=6*, *user_agent=u'pynsq/0.7.0'*, *output_buffer_size=16384*, *output_buffer_timeout=250*, *sample_rate=0*, *io_loop=None*, *auth_secret=None*, *msg_timeout=None*)

Low level object representing a TCP connection to nsqd.

When a message on this connection is requeued and the requeue delay has not been specified, it calculates the delay automatically by an increasing multiple of `requeue_delay`.

Generates the following events that can be listened to with *`nsq.AsyncConn.on()`*:

- `connect`
- `close`
- `error`
- `identify`
- `identify_response`
- `auth`
- `auth_response`
- `heartbeat`
- `ready`
- `message`
- `response`
- `backoff`
- `resume`

Parameters

- **`host`** – the host to connect to
- **`port`** – the post to connect to
- **`timeout`** – the timeout for read/write operations (in seconds)
- **`heartbeat_interval`** – the amount of time (in seconds) to negotiate with the connected producers to send heartbeats (requires nsqd 0.2.19+)

- **`requeue_delay`** – the base multiple used when calculating requeue delay (multiplied by # of attempts)
- **`tls_v1`** – enable TLS v1 encryption (requires nsqd 0.2.22+)
- **`tls_options`** – dictionary of options to pass to ssl.wrap_socket() as `**kwargs`
- **`snappy`** – enable Snappy stream compression (requires nsqd 0.2.23+)
- **`deflate`** – enable deflate stream compression (requires nsqd 0.2.23+)
- **`deflate_level`** – configure the deflate compression level for this connection (requires nsqd 0.2.23+)
- **`output_buffer_size`** – size of the buffer (in bytes) used by nsqd for buffering writes to this connection
- **`output_buffer_timeout`** – timeout (in ms) used by nsqd before flushing buffered writes (set to 0 to disable). **Warning**: configuring clients with an extremely low (`< 25ms`) `output_buffer_timeout` has a significant effect on `nsqd` CPU usage (particularly with `> 50` clients connected).
- **`sample_rate`** – take only a sample of the messages being sent to the client. Not setting this or setting it to 0 will ensure you get all the messages destined for the client. Sample rate can be greater than 0 or less than 100 and the client will receive that percentage of the message traffic. (requires nsqd 0.2.25+)
- **`user_agent`** – a string identifying the agent for this client in the spirit of HTTP (default: `<client_library_name>/<version>`) (requires nsqd 0.2.25+)
- **`auth_secret`** – a string passed when using nsq auth (requires nsqd 1.0+)
- **`msg_timeout`** – the amount of time (in seconds) that nsqd will wait before considering messages that have been delivered to this consumer timed out (requires nsqd 0.2.28+)

`off`(*name*, *callback*)

Stop listening for the named event via the specified callback.

Parameters

- **`name`** (*`string`*) – the name of the event
- **`callback`** (*`callable`*) – the callback that was originally used

`on`(*name*, *callback*)

Listen for the named event with the specified callback.

Parameters

- **`name`** (*`string`*) – the name of the event
- **`callback`** (*`callable`*) – the callback to execute when the event is triggered

`trigger`(*name*, **args*, ***kwargs*)

Execute the callbacks for the listeners on the specified event with the supplied arguments.

All extra arguments are passed through to each callback.

Parameters **`name`** (*`string`*) – the name of the event

CHAPTER 4

`Message` – an NSQ message

class `nsq.`**`Message`**(*id*, *body*, *timestamp*, *attempts*)

A class representing a message received from `nsqd`.

If you want to perform asynchronous message processing use the *`nsq.Message.enable_async()`* method, pass the message around, and respond using the appropriate instance method.

Generates the following events that can be listened to with *`nsq.Message.on()`*:

- `finish`
- `requeue`
- `touch`

NOTE: A calling a message's *`nsq.Message.finish()`* and *`nsq.Message.requeue()`* methods positively and negatively impact the backoff state, respectively. However, sending the `backoff=False` keyword argument to *`nsq.Message.requeue()`* is considered neutral and will not impact backoff state.

Parameters

- **`id`** (*`string`*) – the ID of the message
- **`body`** (*`string`*) – the raw message body
- **`timestamp`** (*`int`*) – the timestamp the message was produced
- **`attempts`** (*`int`*) – the number of times this message was attempted

Variables

- **`id`** – the ID of the message (from the parameter).
- **`body`** – the raw message body (from the parameter).
- **`timestamp`** – the timestamp the message was produced (from the parameter).
- **`attempts`** – the number of times this message was attempted (from the parameter).

`enable_async`()

Enables asynchronous processing for this message.

`nsq.Reader` will not automatically respond to the message upon return of `message_handler`.

`finish`()

Respond to `nsqd` that you've processed this message successfully (or would like to silently discard it).

`has_responded`()

Returns whether or not this message has been responded to.

`is_async`()
Returns whether or not asynchronous processing has been enabled.

`off`(*name*, *callback*)
Stop listening for the named event via the specified callback.

Parameters

- **`name`** (*`string`*) – the name of the event
- **`callback`** (*`callable`*) – the callback that was originally used

`on`(*name*, *callback*)
Listen for the named event with the specified callback.

Parameters

- **`name`** (*`string`*) – the name of the event
- **`callback`** (*`callable`*) – the callback to execute when the event is triggered

`requeue`(***kwargs*)
Respond to `nsqd` that you've failed to process this message successfully (and would like it to be requeued).

Parameters

- **`backoff`** (*`bool`*) – whether or not *`nsq.Reader`* should apply backoff handling
- **`delay`** (*`int`*) – the amount of time (in seconds) that this message should be delayed if -1 it will be calculated based on # of attempts

`touch`()
Respond to `nsqd` that you need more time to process the message.

`trigger`(*name*, **args*, ***kwargs*)
Execute the callbacks for the listeners on the specified event with the supplied arguments.

All extra arguments are passed through to each callback.

Parameters **`name`** (*`string`*) – the name of the event

CHAPTER 5

LegacyReader – backwards compatible Reader

class nsq.**LegacyReader**(**args*, ***kwargs*)

In v0.5.0 we dropped support for "tasks" in the *nsq.Reader* API in favor of a single message handler.

LegacyReader is a backwards compatible API for clients interacting with v0.5.0+ that want to continue to use "tasks".

Usage:

```
from nsq import LegacyReader as Reader
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index