pynsq Documentation

Release 0.5.0

Matt Reiferson and Jehiah Czebotar

October 17, 2014

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

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.

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

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.

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

touch()

Respond to nsqd that you need more time to process the message.

CHAPTER 2

Consumers

nsq.**run**()
Starts any instantiated nsq.Reader or nsq.Writer

2.1 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*, *requeue_delay=90*, *lookupd_poll_interval=60*, *low_rdy_idle_timeout=10*, *heartbeat_interval=30*, *max_backoff_duration=128*, *lookupd_poll_jitter=0.3*, *tls_v1=False*, *tls_options=None*)

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.

It is responsible for sending FIN or REQ commands based on return value of message_handler. When re-queueing, an increasing delay will be calculated automatically.

Additionally, when message processing fails, it will backoff in increasing multiples of requeue_delay between updating of RDY count.

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
- **requeue_delay** – the base multiple used when re-queueing (multiplied by # of attempts)
- **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
- **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)

- **heartbeat_interval** – the amount of time in seconds to negotiate with the connected producers to send heartbeats (requires nsqd 0.2.19+)
- **max_backoff_duration** – the maximum time we will allow a backoff state to last in seconds
- **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.
- **tls_v1** – enable TLS v1 encryption (requires nsqd 0.2.22+)
- **tls_options** – dictionary of options to pass to ssl.wrap_socket() as **kwargs

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

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.).

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_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

CHAPTER 3

Producers

nsq.**run**()
Starts any instantiated nsq.Reader or nsq.Writer

3.1 Writer – high-level producer

class nsq.**Writer**(*nsqd_tcp_addresses*, *heartbeat_interval=30*)
A high-level producer class built on top of the Tornado IOLoop supporting async publishing (PUB & MPUB) 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

        # 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 of (addresses, port) of the `nsqd` instances this writer should publish to
- **heartbeat_interval** – the interval in seconds to configure heartbeats w/ `nsqd`

CHAPTER 4

`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 5

Indices and tables

- *genindex*
- *modindex*
- *search*