gnsq Documentation

Release 0.3.0

Trevor Olson

June 14, 2015

Contents

CHAPTER 1

gnsq

A gevent based python client for NSQ distributed messaging platform.

Features include:

- Free software: BSD license
- Documentation: https://gnsq.readthedocs.org
- Battle tested on billions and billions of messages *</sagan>*
- Based on gevent for fast concurrent networking
- Fast and flexible signals with Blinker
- Automatic nsqlookupd discovery and back-off
- Support for TLS, DEFLATE, and Snappy
- Full HTTP clients for both nsqd and nsqlookupd

1.1 Installation

At the command line:

```
$ easy_install gnsq
```

Or even better, if you have virtualenvwrapper installed:

```
$ mkvirtualenv gnsq
$ pip install gnsq
```

Currently there is support for Python 2.6 and Python 2.7. Support for Python 3 is dependent on gevent support.

1.2 Usage

First make sure nsq is installed and running. Next create a nsqd connection and publish some messages to your topic:

```
import gnsq
conn = gnsq.Nsqd(address='localhost', http_port=4151)

conn.publish('topic', 'hello gevent!')
conn.publish('topic', 'hello nsq!')
```

Then create a Reader to consume messages from your topic:

```
reader = gnsq.Reader('topic', 'channel', 'localhost:4150')

@reader.on_message.connect
def handler(reader, message):
    print 'got message:', message.body

reader.start()
```

1.3 Dependencies

Optional snappy support depends on the *python-snappy* package which in turn depends on libsnappy:

```
# Debian
$ sudo apt-get install libsnappy-dev

# Or OS X
$ brew install snappy

# And then install python-snappy
$ pip install python-snappy
```

1.4 Contributing

Feedback, issues, and contributions are always gratefully welcomed. See the contributing guide for details on how to help and setup a development environment.

CHAPTER 2

Contents

2.1 Reader: high-level consumer

class gnsq.**Reader**(*topic*, *channel*, *nsqd_tcp_addresses=[]*, *lookupd_http_addresses=[]*, *name=None*, *message_handler=None*, *async=False*, *max_tries=5*, *max_in_flight=1*, *max_concurrency=0*, *requeue_delay=0*, *lookupd_poll_interval=60*, *lookupd_poll_jitter=0.3*, *low_ready_idle_timeout=10*, *max_backoff_duration=128*, *backoff_on_requeue=True*, ***kwargs*)

High level NSQ consumer.

A Reader will connect to the nsqd tcp addresses or poll the provided nsqlookupd http addresses for the configured topic and send signals to message handlers connected to the *on_message* signal or provided by *message_handler*.

Messages will automatically be finished when the message handle returns unless the readers *async* flag is set to *True*. If an exception occurs or `gnsq.errors.NSQRequeueMessage` is raised, the message will be requeued.

The Reader will handle backing off of failed messages up to a configurable *max_interval* as well as automatically reconnecting to dropped connections.

Parameters

- **topic** – specifies the desired NSQ topic
- **channel** – specifies the desired NSQ 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
- **name** – a string that is used for logging messages (defaults to 'gnsq.reader.topic.channel')
- **message_handler** – the callable that will be executed for each message received
- **async** – consider the message handling to be async. The message will not automatically be finished after the handler returns and must manually be called
- **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

- **max_concurrency** – the maximum number of messages that will be handled concurrently. Defaults to the number of nsqd connections. Setting *max_concurrency* to *-1* will use the systems cpu count.
- **requeue_delay** – the default delay to use when requeueing a failed message
- **lookupd_poll_interval** – the amount of time in seconds between querying all of the supplied nsqlookupd instances. A random amount of time based on this 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.
- **low_ready_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. If zero, backoff wil not occur
- **backoff_on_requeue** – if False, backoff will only occur on exception
- ****kwargs** – passed to *gnsq.Nsqd* initialization

close()
Immediately close all connections and stop workers.

is_running
Check if reader is currently running.

is_starved
Evaluate whether any of the connections are starved.

This property should be used by message handlers to reliably identify when to process a batch of messages.

join(*timeout=None*, *raise_error=False*)
Block until all connections have closed and workers stopped.

on_auth
Emitted after a connection is successfully authenticated.

The signal sender is the reader and the *conn* and parsed *response* are sent as arguments.

on_close
Emitted after `close()`.

The signal sender is the reader.

on_error
Emitted when an error is received.

The signal sender is the reader and the *error* is sent as an argument.

on_exception
Emitted when an exception is caught while handling a message.

The signal sender is the reader and the *message* and *error* are sent as arguments.

on_finish
Emitted after a message is successfully finished.

The signal sender is the reader and the *message_id* is sent as an argument.

on_giving_up
Emitted after a giving up on a message.

Emitted when a message has exceeded the maximum number of attempts (*max_tries*) and will no longer be requeued. This is useful to perform tasks such as writing to disk, collecting statistics etc. The signal sender is the reader and the *message* is sent as an argument.

on_message
Emitted when a message is received.

The signal sender is the reader and the *message* is sent as an argument. The *message_handler* param is connected to this signal.

on_requeue
Emitted after a message is requeued.

The signal sender is the reader and the *message_id* and *timeout* are sent as arguments.

on_response
Emitted when a response is received.

The signal sender is the reader and the *response* is sent as an argument.

publish(*topic*, *message*)
Publish a message to a random connection.

start(*block=True*)
Start discovering and listing to connections.

2.2 Nsqd client

class gnsq.**Nsqd**(*address='127.0.0.1'*, *tcp_port=4150*, *http_port=4151*, *timeout=60.0*, *client_id=None*, *hostname=None*, *heartbeat_interval=30*, *output_buffer_size=16384*, *output_buffer_timeout=250*, *tls_v1=False*, *tls_options=None*, *snappy=False*, *deflate=False*, *deflate_level=6*, *sample_rate=0*, *auth_secret=None*, *user_agent='gnsq/0.3.0'*)
Low level object representing a TCP or HTTP connection to nsqd.

Parameters

- **address** – the host or ip address of the nsqd
- **tcp_port** – the nsqd tcp port to connect to
- **http_port** – the nsqd http port to connect to
- **timeout** – the timeout for read/write operations (in seconds)
- **client_id** – an identifier used to disambiguate this client (defaults to the first part of the hostname)
- **hostname** – the hostname where the client is deployed (defaults to the clients hostname)
- **heartbeat_interval** – the amount of time in seconds to negotiate with the connected producers to send heartbeats (requires nsqd 0.2.19+)
- **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).

- **tls_v1** – enable TLS v1 encryption (requires nsqd 0.2.22+)
- **tls_options** – dictionary of options to pass to ssl.wrap_socket()
- **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+)
- **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+)
- **auth_secret** – a string passed when using nsq auth (requires nsqd 0.2.29+)
- **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()
Send authorization secret to nsqd.

close()
Indicate no more messages should be sent.

close_stream()
Close the underlying socket.

connect()
Initialize connection to the nsqd.

create_channel(*topic*, *channel*)
Create a channel for an existing topic.

create_topic(*topic*)
Create a topic.

delete_channel(*topic*, *channel*)
Delete an existing channel for an existing topic.

delete_topic(*topic*)
Delete a topic.

empty_channel(*topic*, *channel*)
Empty all the queued messages for an existing channel.

empty_topic(*topic*)
Empty all the queued messages for an existing topic.

finish(*message_id*)
Finish a message (indicate successful processing).

identify()
Update client metadata on the server and negotiate features.

Returns nsqd response data if there was feature negotiation, otherwise *None*

info()
Returns version information.

is_connected
Check if the client is currently connected.

is_starved
Evaluate whether the connection is starved.

This property should be used by message handlers to reliably identify when to process a batch of messages.

listen()
Listen to incoming responses until the connection closes.

multipublish(*topic*, *messages*)
Publish an iterable of messages in one roundtrip.

If connected, the messages will be sent over tcp. Otherwise it will fall back to http.

multipublish_http(*topic*, *messages*)
Publish an iterable of messages to the given topic over http.

multipublish_tcp(*topic*, *messages*)
Publish an iterable of messages to the given topic over tcp.

nop()
Send no-op to nsqd. Used to keep connection alive.

on_auth
Emitted after the connection is successfully authenticated.

The signal sender is the connection and the parsed *response* is sent as arguments.

on_close
Emitted after `close_stream()`.

Sent after the connection socket has closed. The signal sender is the connection.

on_error
Emitted when an error frame is received.

The signal sender is the connection and the *error* is sent as an argument.

on_finish
Emitted after `finish()`.

Sent after a message owned by this connection is successfully finished. The signal sender is the connection and the *message_id* is sent as an argument.

on_message
Emitted when a message frame is received.

The signal sender is the connection and the *message* is sent as an argument.

on_requeue
Emitted after `requeue()`.

Sent after a message owned by this connection is requeued. The signal sender is the connection and the *message_id*, *timeout* and *backoff* flag are sent as arguments.

on_response
Emitted when a response frame is received.

The signal sender is the connection and the *response* is sent as an argument.

pause_channel(*topic*, *channel*)
Pause message flow to all channels on an existing topic.

Messages will queue at topic.

ping()
Monitoring endpoint.

Returns should return *"OK"*, otherwise raises an exception.

publish(*topic*, *data*)
Publish a message.

If connected, the message will be sent over tcp. Otherwise it will fall back to http.

publish_http(*topic*, *data*)
Publish a message to the given topic over http.

publish_tcp(*topic*, *data*)
Publish a message to the given topic over tcp.

read_response()
Read an individual response from nsqd.

Returns tuple of the frame type and the processed data.

ready(*count*)
Indicate you are ready to receive *count* messages.

requeue(*message_id*, *timeout=0*, *backoff=True*)
Re-queue a message (indicate failure to process).

stats()
Return internal instrumented statistics.

subscribe(*topic*, *channel*)
Subscribe to a nsq *topic* and *channel*.

touch(*message_id*)
Reset the timeout for an in-flight message.

unpause_channel(*topic*, *channel*)
Resume message flow to channels of an existing, paused, topic.

2.3 Nsqlookupd client

class gnsq.**Lookupd**(*address='http://localhost:4161/'*)
Low level client for nsqlookupd.

Parameters **address** – nsqlookupd http address (default: http://localhost:4161/)

channels(*topic*)
Returns all known channels of a topic.

delete_channel(*topic*, *channel*)
Deletes an existing channel of an existing topic.

delete_topic(*topic*)
Deletes an existing topic.

info()
Returns version information.

lookup(*topic*)
Returns producers for a topic.

nodes()
Returns all known nsqd.

ping()
Monitoring endpoint.

Returns should return *"OK"*, otherwise raises an exception.

tombstone_topic_producer(*topic*, *node*)
Tombstones a specific producer of an existing topic.

topics()
Returns all known topics.

2.4 NSQ Message

class gnsq.**Message**(*timestamp*, *attempts*, *id*, *body*)
A class representing a message received from nsqd.

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.

on_finish
Emitted after `finish()`.

The signal sender is the message instance.

on_requeue
Emitted after `requeue()`.

The signal sender is the message instance and sends the *timeout* and a *backoff* flag as arguments.

on_touch
Emitted after `touch()`.

The signal sender is the message instance.

requeue(*time_ms=0*, *backoff=True*)
Respond to nsqd that you've failed to process this message successfully (and would like it to be requeued).

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

2.5 Signals

Both Reader and Nsqd classes expose various signals provided by the Blinker library.

2.5.1 Subscribing to signals

To subscribe to a signal, you can use the `connect()` method of a signal. The first argument is the function that should be called when the signal is emitted, the optional second argument specifies a sender. To unsubscribe from a signal, you can use the `disconnect()` method.

```
def error_handler(reader, error):
    print 'Got an error:', error

reader.on_error.connect(error_handler)
```

You can also easily subscribe to signals by using `connect()` as a decorator:

```
@reader.on_giving_up.connect
def handle_giving_up(reader, message):
    print 'Giving up on:', message.id
```

2.6 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.6.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/wtolson/gnsq/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

gnsq could always use more documentation, whether as part of the official gnsq docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/wtolson/gnsq/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.6.2 Get Started!

Ready to contribute? Here's how to set up *gnsq* for local development.

1. Fork the *gnsq* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/gnsq.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper and libsnappy installed, this is how you set up your fork for local development:

```
$ mkvirtualenv gnsq
$ cd gnsq/
$ pip install -r requirements.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 gnsq tests
$ py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6 and 2.7. Check https://travis-ci.org/wtolson/gnsq/pull_requests and make sure that the tests pass for all supported Python versions.

2.6.4 Tips

To run a subset of tests:

```
$ py.test tests/test_basic.py
```

2.7 Credits

2.7.1 Development Lead

- Trevor Olson <trevor@heytrevor.com>

2.7.2 Contributors

None yet. Why not be the first?

2.8 History

2.8.1 0.3.0 (2015-06-14)

- Fix extra backoff success/failures during backoff period.
- Fix case where handle_backoff is never called.
- Add backoff parameter to message.requeue().
- Allow overriding backoff on NSQRequeueMessage error.
- Handle connection failures while starting/completing backoff.

2.8.2 0.2.3 (2015-02-16)

- Remove disconnected nsqd messages from the worker queue.
- #4 - Fix crash in Reader.random_ready_conn (thanks @ianpreston).

2.8.3 0.2.2 (2015-01-12)

- Allow finishing and requeuing in sync handlers.

2.8.4 0.2.1 (2015-01-12)

- Topics and channels are now valid to 64 characters.
- Ephemeral topics are now valid.
- Adjustable backoff behaviour.

2.8.5 0.2.0 (2014-08-03)

- Warn on connection failure.
- Add extra requires for snappy.
- Add support for nsq auth protocal.

2.8.6 0.1.4 (2014-07-24)

- Preemptively update ready count.
- Dependency and contributing documentation.
- Support for nsq back to 0.2.24.

2.8.7 0.1.3 (2014-07-08)

- Block as expected on start, even if already started.
- Raise runtime error if starting the reader without a message handler.
- Add on_close signal to the reader.
- Allow upgrading to tls+snappy or tls+deflate.

2.8.8 0.1.2 (2014-07-08)

- Flush delfate buffer for each message.

2.8.9 0.1.1 (2014-07-07)

- Fix packaging stream submodule.
- Send queued messages before closing socket.
- Continue to read from socket on EAGAIN

2.8.10 0.1.0 (2014-07-07)

- First release on PyPI.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

R

S

T

U