Stomp Documentation

Release 4.1.22

Jason R Briggs

Dec 23, 2018

Contents

Stomp.py was created by Jason R Briggs. You can view outstanding issues, and find further info, on the Github project page.

Contents:

CHAPTER 1

Quick start

Stomp.py is a Python library providing access to a message broker using the STOMP protocol - either programmatically or using a command line client.

1.1 Stomp.py API

```
>>> import time
>>> import sys
>>>
>>> import stomp
>>>
>>> class MyListener(stomp.ConnectionListener):
>>>     def on_error(self, headers, message):
>>>         print('received an error "%s"' % message)
>>>     def on_message(self, headers, message):
>>>         print('received a message "%s"' % message)

>>> conn = stomp.Connection()
>>> conn.set_listener('', MyListener())
>>> conn.start()
>>> conn.connect('admin', 'password', wait=True)
>>>
>>> conn.subscribe(destination='/queue/test', id=1, ack='auto')
>>>
>>> conn.send(body=' '.join(sys.argv[1:]), destination='/queue/test')
>>>
>>> time.sleep(2)
>>> conn.disconnect()
```

1.2 Command-line Client

Assuming stomp.py is installed (using pip) in the site-packages directory (e.g. lib/python3.3/site-packages), hereby referred to as ${SITEPACKAGES}, then you can run the command line client as follows:

```
python ${SITEPACKAGES}/stomp -H localhost -P 61613
```

As of version 4.0.3, a stomp.py is also installed into the bin dir (at least on unix), so you can also run:

```
stomp -H localhost -P 61613
```

After a successful connection, you can type commands such as:

```
subscribe /queue/test
send /queue/test hello world
```

If you need to pass a username and password to the client:

```
python ${SITEPACKAGES}/stomp -H localhost -P 61613 -U admin -W password
stomp -H localhost -P 61613 -U admin -W password
```

Type help for more information once you're running the command-line interface, or run the following to see the list of startup arguments:

```
python ${SITEPACKAGES}/stomp --help
stomp --help
```

CHAPTER 2

Introduction to Stomp.py

2.1 About Stomp.py

Stomp.py started as an "itch-scratching" project, after discovering that the message broker we were using for inter-application communications in a telecommunications platform, had a text-based protocol called STOMP you could use for access. We wanted something which could randomly send a variation of messages, easily scriptable - and there was only one other Python-based client library available at the time (which didn't work, and it looked as if the project had stalled). So after a number of evenings spent coding (ah! those were the days - when one could get away with endless coding in the evenings), the first version of stomp.py was created (supporting the basics of the 1.0 protocol, a smidgen of a CLI, and little else). In the 8 or 9 years since its inception, support for the subsequent versions of STOMP have been added, and the command line client has been significantly enhanced.

- Stomp.py currently supports all versions of the stomp protocol (1.0, 1.1 and 1.2)
- Both Python 2 and Python 3 are supported
- The command-line client is installed via pip and has a number of useful features for testing
- The code is perfunctorily tested on: ActiveMQ, RabbitMQ, stompserver, and has been reported to work with JBossMessaging in the distant past. Full test suite runs against Apache Apollo (for info on setting up VirtualBox for testing, contact the developer).

2.2 Getting Help

View outstanding issues on the GitHub issues list, or raise a request for help (note that stomp.py is 'intermittently' supported at times).

2.3 Contributors

Julian Scheid (Rising Sun Pictures)

Andreas Schobel

Fernando Ciciliati

Eugene Strulyov

Gavin M. Roy

Martin Pieuchot

Joe Gdaniec

Jayson Vantuyl

Tatiana Al-Chueyr Martins

Rafael Durán Casteñada

Chaskiel Grundman

Ville Skyttä

CHAPTER 3

Using the API

3.1 Establishing a connection

The simplest way to establish a connection, assuming the message broker is running on the local machine, is:

```
>>> import stomp
>>> c = stomp.Connection([('127.0.0.1', 62613)])
>>> c.start()
>>> c.connect('admin', 'password', wait=True)
```

By default this represents a STOMP 1.1 connection. You can request a specific version of the connection using one of the following:

```
>>> c = stomp.Connection10([('127.0.0.1', 62613)])
>>> c = stomp.Connection11([('127.0.0.1', 62613)])
>>> c = stomp.Connection12([('127.0.0.1', 62613)])
```

The first parameter to a `Connection` is `host_and_ports`. This is a list of tuples, each containing ip address (which could be an ipv6 address) and the port where the message broker is listening for stomp connections. The general idea with the list is to try each address until a successful socket connection is established (giving the ability to provide multiple brokers for failover).

An example of setting up a connection with failover addresses might be:

```
>>> import stomp
>>> c = stomp.Connection([('192.168.1.100', 61613), ('192.168.1.101', 62613)])
```

And here's an example of an ipv6 connection:

```
>>> import stomp
>>> c = stomp.Connection(['fe80::a00:27ff:fe90:3f1a%en1', 62613])
```

There are a number of other parameters for initialising the connection (looking at the StompConnection12 class):

class stomp.connect.**StompConnection12**(*host_and_ports=None*, *prefer_localhost=True*, *try_loopback_connect=True*, *reconnect_sleep_initial=0.1*, *reconnect_sleep_increase=0.5*, *reconnect_sleep_jitter=0.1*, *reconnect_sleep_max=60.0*, *reconnect_attempts_max=3*, *use_ssl=False*, *ssl_key_file=None*, *ssl_cert_file=None*, *ssl_ca_certs=None*, *ssl_cert_validator=None*, *wait_on_receipt=False*, *ssl_version=3*, *timeout=None*, *heartbeats=(0, 0)*, *keepalive=None*, *vhost=None*, *auto_decode=True*, *encoding='utf-8'*, *auto_content_length=True*, *heart_beat_receive_scale=1.5*, *recv_bytes=1024*)

Represents a 1.2 connection (comprising transport plus 1.2 protocol class). See `stomp.transport.Transport` for details on the initialisation parameters.

The `start` method is next - which performs the actual socket connection to the remote server and starts a separate receiver thread. The final step is to call the `connect` method (corresponding to the CONNECT frame):

Protocol12.**connect**(*username=None*, *passcode=None*, *wait=False*, *headers=None*, ***keyword_headers*)

Send a STOMP CONNECT frame. Differs from 1.0 and 1.1 versions in that the HOST header is enforced.

Parameters

- **username** (*str*) – optionally specify the login user
- **passcode** (*str*) – optionally specify the user password
- **wait** (*bool*) – wait for the connection to complete before returning
- **headers** (*dict*) – a map of any additional headers to send with the subscription
- **keyword_headers** – any additional headers to send with the subscription

Note that connect also allows for a map (dict) of headers to be provided, and will merge these with any additional named parameters to build the headers for the STOMP frame, allowing for non-standard headers to be transmitted to the broker.

3.2 Sending and receiving messages

Once the connection is established, you can send messages using the `send` method:

Protocol12.**send**(*destination*, *body*, *content_type=None*, *headers=None*, ***keyword_headers*)

Send a message to a destination in the messaging system (as per https://stomp.github.io/stomp-specification-1.2.html#SEND)

Parameters

- **destination** (*str*) – the destination (such as a message queue - for example '/queue/test' - or a message topic)

- **body** – the content of the message
- **content_type** (*str*) – the MIME type of message
- **headers** (*dict*) – additional headers to send in the message frame
- **keyword_headers** – any additional headers the broker requires

To receive messages back from the messaging system, you need to setup some sort of listener on your connection, and then subscribe to the destination (see STOMP subscribe). Listeners are simply a subclass which implements the methods in the ConnectionListener class (see this page for more detail). Stomp provides a few implementations of listeners, but the simplest is `PrintingListener` which just prints all interactions between the client and server. A simple example of this in action is:

```
>>> from stomp import *
>>> c = Connection([('127.0.0.1', 62613)])
>>> c.set_listener('', PrintingListener())
>>> c.start()
>>> c.connect('admin', 'password', wait=True)
on_connecting 127.0.0.1 62613
on_send STOMP {'passcode': 'password', 'login': 'admin', 'accept-version': '1.2',
↪'host': '127.0.0.1'}
on_connected {'server': 'apache-apollo/1.7.1', 'host-id': 'mybroker', 'session':
↪'mybroker-13e0', 'heart-beat': '100,10000', 'version': '1.2', 'user-id': 'admin'}
>>> c.subscribe('/queue/test', 123)
on_send SUBSCRIBE {'id': 123, 'ack': 'auto', 'destination': '/queue/test'}
>>> c.send('/queue/test', 'a test message')
on_send SEND {'content-length': 5, 'destination': '/queue/test'} b'a test message'
on_before_message {'destination': '/queue/test', 'message-id': 'mybroker-13e01',
↪'subscription': '123', 'ack': '2', 'content-length': '5'} a test message
on_message {'destination': '/queue/test', 'message-id': 'mybroker-13e01',
↪'subscription': '123', 'ack': '2', 'content-length': '5'} a test message
```

You can see the responses from the message system in the `on_connected`, and `on_message` output. The stomp frames sent to the server can be seen in each `on_send` output (an initial STOMP connect frame, SUBSCRIBE and then SEND).

In the case of the subscribe method, as of STOMP 1.1, the `id` parameter is required (if connecting with STOMP 1.0, only the destination is required):

`Protocol12.`**`subscribe`**(*destination*, *id*, *ack='auto'*, *headers=None*, ***keyword_headers*)

Subscribe to a destination

Parameters

- **destination** (*str*) – the topic or queue to subscribe to
- **id** (*str*) – the identifier to uniquely identify the subscription
- **ack** (*str*) – either auto, client or client-individual (see https://stomp.github.io/stomp-specification-1.2.html#SUBSCRIBE for more info)
- **headers** (*dict*) – a map of any additional headers to send with the subscription
- **keyword_headers** – any additional headers to send with the subscription

Note that listeners can be named so you can use more that one type of listener at the same time:

```
>>> c.set_listener('stats', StatsListener())
>>> c.set_listener('print', PrintingListener())
```

You unsubscribe from a topic or queue using the unique `id` for the subscription:

```
>>> c.subscribe('/queue/test', 123)
>>> c.unsubscribe(123)
```

3.3 Acks and Nacks

Acknowledgements are a way to tell the message server that a message was either consumed, or not. Assume a collection of clients on a server listening on a queue, and a message which requires significant processing. One of the clients receives the message, checks resource usage on the server and decides to send a nack as a consequence. The message server could, at that point, decide to send to a failover server for processing (that's a possible use, anyway).

Use the client or client-individual acknowledgement parameter (see here for a description) with the subscription, in order to use acks and nacks. Afterwards, you use the message and subscription ids to ack or nack the message:

```
>>> conn.subscribe('/queue/test', id=4, ack='client')
on_before_message {'message-id': 'mybroker-14aa2', 'destination': '/queue/test',
↪'subscription': '4', 'content-length': '14'} test message 1
on_message {'message-id': 'mybroker-14aa2', 'destination': '/queue/test',
↪'subscription': '4', 'content-length': '14'} test message 1
>>> conn.ack('mybroker-14aa2', 4)

on_before_message {'message-id': 'mybroker-14ab2', 'destination': '/queue/test',
↪'subscription': '4', 'content-length': '14'} test message 2
on_message {'message-id': 'mybroker-14ab2', 'destination': '/queue/test',
↪'subscription': '4', 'content-length': '14'} test message 2
>>> conn.nack('mybroker-14ab2', 4)
```

3.4 Transactions

The STOMP protocol provides a way to transmit messages to a broker inside a transaction, which are held on the server until the transaction is either committed - at which point they're sent - or aborted - where the messages are discarded.

Begin a transaction using the `begin` method, which returns the transaction id you then use when sending messages (you can also generate your own transaction id and pass that as a parameter to `begin`):

```
>>> conn.subscribe('/queue/test', id=5)
>>> txid = conn.begin()
>>> conn.send('/queue/test', 'test1', transaction=txid)
>>> conn.send('/queue/test', 'test2', transaction=txid)
>>> conn.send('/queue/test', 'test3', transaction=txid)
>>> conn.commit(txid)

on_message {'subscription': '5', 'content-length': '5', 'destination': '/queue/test',
↪'message-id': 'mybroker-14b03', 'transaction': 'b39f3136-46a3-4e11-8ba8-845e36d48412
↪'} test1
on_message {'subscription': '5', 'content-length': '5', 'destination': '/queue/test',
↪'message-id': 'mybroker-14b04', 'transaction': 'b39f3136-46a3-4e11-8ba8-845e36d48412
↪'} test2
on_message {'subscription': '5', 'content-length': '5', 'destination': '/queue/test',
↪'message-id': 'mybroker-14b05', 'transaction': 'b39f3136-46a3-4e11-8ba8-845e36d48412
↪'} test3
```

Abort a transaction (and discard the sent messages using `abort`):

```
>>> conn.subscribe('/queue/test', id=6)
>>> txid = conn.begin()
>>> conn.send('/queue/test', 'test4', transaction=txid)
>>> conn.send('/queue/test', 'test5', transaction=txid)
>>> conn.abort(txid)
```

3.5 Disconnect

Stomp.py supports graceful shutdown/disconnections through a receipt parameter (automatically generated if you don't provide it). The connection is only dropped when the server sends back a response to that receipt:

```
>>> conn.disconnect()
on_send DISCONNECT {'receipt': '825a5cd6-9e3c-4a72-8051-72348a94f5ce'}
on_receipt {'receipt-id': '825a5cd6-9e3c-4a72-8051-72348a94f5ce'}
on_disconnected
```

3.6 Dealing with disconnects

You can use a listener to deal with connection failures, and gracefully reconnect. Consider the below 'server' code:

```
import os
import time
import stomp

def connect_and_subscribe(conn):
    conn.start()
    conn.connect('guest', 'guest', wait=True)
    conn.subscribe(destination='/queue/test', id=1, ack='auto')

class MyListener(stomp.ConnectionListener):
    def __init__(self, conn):
        self.conn = conn

    def on_error(self, headers, message):
        print('received an error "%s"' % message)

    def on_message(self, headers, message):
        print('received a message "%s"' % message)
        for x in range(10):
            print(x)
            time.sleep(1)
        print('processed message')

    def on_disconnected(self):
        print('disconnected')
        connect_and_subscribe(self.conn)

conn = stomp.Connection([('localhost', 62613)], heartbeats=(4000, 4000))
conn.set_listener('', MyListener(conn))
connect_and_subscribe(conn)
time.sleep(60)
conn.disconnect()
```

The listener in this code has an, arguably broken, message handler (on_message) which takes longer to process than the heartbeat time of 4 seconds (4000); resulting in a heartbeat timeout when a message is received, and a subsequent disconnect. The on_disconnected method then reconnects and continues processing. You can test the results of this by running the above code, and sending a message using the following 'client':

```
import stomp
conn = stomp.Connection([('localhost', 62613)])
conn.start()
conn.connect('guest', 'guest', wait=True)
conn.send('/queue/test', 'test message')
```

CHAPTER 4

Using the Command-line client application

Once stomp.py is installed, access the command-line client as follows:

```
stomp -H 127.0.0.1 -P 61613 -U admin -W password
```

Find more info using –help:

```
$ stomp --help
Usage: stomp [options]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -H HOST, --host=HOST  Hostname or IP to connect to. Defaults to localhost if
                        not specified.
  -P PORT, --port=PORT  Port providing stomp protocol connections. Defaults to
                        61613 if not specified.
  -U USER, --user=USER  Username for the connection
  -W PASSWORD, --password=PASSWORD
                        Password for the connection
  -F FILENAME, --file=FILENAME
                        File containing commands to be executed, instead of
                        prompting from the command prompt.
  -S STOMP, --stomp=STOMP
                        Set the STOMP protocol version.
  -L LISTEN, --listen=LISTEN
                        Listen for messages on a queue/destination
  -V VERBOSE, --verbose=VERBOSE
                        Verbose logging "on" or "off" (if on, full headers
                        from stomp server responses are printed)
  --ssl                 Enable SSL connection
```

And you can also get more help within the application using the help command:

```
> help

Documented commands (type help <topic>):
========================================
EOF    begin   help  rollback  sendfile   stats        ver
abort  commit  nack  run       sendrec    subscribe    version
ack    exit    quit  send      sendreply  unsubscribe
```

Some of the differences to the programmatic API are the ability to run a script, and to send files (`run`, `sendfile`, `stats`):

```
> help run
Usage:
    run <filename>

Description:
    Execute commands in a specified file

> help sendfile
Usage:
    sendfile <destination> <filename>

Required Parameters:
    destination - where to send the message
    filename - the file to send

Description:
    Sends a file to a destination in the messaging system.

> help stats
Usage:
    stats [on|off]

Description:
    Record statistics on messages sent, received, errors, etc. If no argument␣
↪(on|off) is specified,
    dump the current statistics.
```

Apart from that, the commands are largely inline with what you can do programmatically. Note that you can run it as a normal CLI, as a standalone listener and use it to run a script of commands.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index