MoaT-MQTT Documentation

Release 0.10

Nicolas Jouanin

Sep 19, 2022

Contents

`MoaT-MQTT` is an open source MQTT client and broker implementation.

Built on top of `asyncio`, Python's standard asynchronous I/O framework, MoaT-MQTT provides a straightforward API based on coroutines, making it easy to write highly concurrent applications.

CHAPTER 1

Features

MoaT-MQTT implements the full set of MQTT 3.1.1 protocol specifications and provides the following features:

- Support QoS 0, QoS 1 and QoS 2 messages flow
- Client auto-reconnection on network lost
- Authentication through password file (more methods can be added through a plugin system)
- Basic `$SYS` topics
- TCP and websocket support
- SSL support over TCP and websocket
- Plugin system

CHAPTER 2

Requirements

MoaT-MQTT is written in asynchronous Python, based on the `anyio` library.

CHAPTER 3

Installation

It is not recommended to install third-party library in Python system packages directory. The preferred way for installing MoaT-MQTT is to create a virtual environment and then install all the dependencies you need. Refer to PEP 405 to learn more.

Once you have a environment setup and ready, MoaT-MQTT can be installed with the following command

```
(venv) $ pip install moat-mqtt
```

`pip` will download and install MoaT-MQTT and all its dependencies.

CHAPTER 4

User guide

If you need MoaT-MQTT for running a MQTT client or deploying a MQTT broker, the *Quickstart* describes how to use console scripts provided by MoaT-MQTT.

If you want to develop an application which needs to connect to a MQTT broker, the *MQTTClient API* documentation explains how to use MoaT-MQTT API for connecting, publishing and subscribing with a MQTT broker.

If you want to run you own MQTT broker, th *Broker API reference* reference documentation explains how to embed a MQTT broker inside a Python application.

News and updates are listed in the *Changelog*.

4.1 Quickstart

A quick way for getting started with `MoaT-MQTT` is to use console scripts provided for :

- publishing a message on some topic on an external MQTT broker.
- subscribing some topics and getting published messages.
- running an autonomous MQTT broker

4.1.1 Installation

That's easy:

```
(venv) $ pip install moat-mqtt
```

4.1.2 Sample code

As `MoaT-MQTT` is async Python, you need to wrap all examples with:

```
async def main():
   [ actual sample code here ]
anyio.run(main)
```

The easiest way to do this is to use the `asyncclick` package:

```
import asyncclick as click
@click.command()
@click.option('-t','--test',is_flag=True, help="Set Me")
async def main(test):
   if not test:
      raise click.UsageError("I told you to set me")  # :-)
   [ actual sample code here ]

main()  # click.command() wraps that in a call to anyio.run()
```

Connecting to a broker

An MQTT connection is typically used as a context manager:

```
async with open_mqttclient(uri='mqtt://localhost:1883', codec='utf8') as C:
   await some_mqtt_commands()
```

Sending messages

That's easy:

```
async with open_mqttclient(...) as C:
   async C.publish("one/two/three/four", [1,2,3,4], codec="msgpack")
```

Receiving messages

Receiving uses another context manager:

```
async with open_mqttclient(...) as C:
   async with C.subscription("one/two/#", codec="msgpack") as S:
      async for msg in S:
         print("I got",msg)
```

The subscription affords a `publish` method which inherits its codec and QoS settings.

If you want to process multiple subscriptions in parallel, the easiest way is to use multiple tasks.

4.1.3 Console scripts

A quick way for getting started with `MoaT-MQTT` is to examine the code in `MoaT-MQTT`'s console scripts.

These scripts are installed automatically when installing `MoaT-MQTT`.

Publishing messages

`moat mqtt pub` is a command-line tool which can be used for publishing some messages on a topic. It requires a few arguments like broker URL, topic name, QoS and data to send. Additional options allow more complex use case.

Publishing `` `some_data `` to a `/test` topic on is as simple as :

```
$ moat mqtt pub --url mqtt://test.mosquitto.org -t /test -m some_data
[2015-11-06 22:21:55,108] :: INFO - pub/5135-MacBook-Pro.local Connecting to broker
[2015-11-06 22:21:55,333] :: INFO - pub/5135-MacBook-Pro.local Publishing to '/test'
[2015-11-06 22:21:55,336] :: INFO - pub/5135-MacBook-Pro.local Disconnected from␣
↪broker
```

This will use insecure TCP connection to connect to test.mosquitto.org. `moat mqtt pub` also allows websockets and secure connection:

```
$ moat mqtt pub --url ws://test.mosquitto.org:8080 -t /test -m some_data
[2015-11-06 22:22:42,542] :: INFO - pub/5157-MacBook-Pro.local Connecting to broker
[2015-11-06 22:22:42,924] :: INFO - pub/5157-MacBook-Pro.local Publishing to '/test'
[2015-11-06 22:22:52,926] :: INFO - pub/5157-MacBook-Pro.local Disconnected from␣
↪broker
```

`moat mqtt pub` can read from file or stdin and use data read as message payload:

```
$ some_command | moat mqtt pub --url mqtt://localhost -t /test -l
```

See references/moat_mqtt_pub reference documentation for details about available options and settings.

Subscribing a topic

`moat mqtt sub` is a command-line tool which can be used to subscribe for some pattern(s) on a broker and get date from messages published on topics matching these patterns by other MQTT clients.

Subscribing a `/test/#` topic pattern is done with :

```
$ moat mqtt sub --url mqtt://localhost -t /test/#
```

This command will run forever and print on the standard output every messages received from the broker. The `-n` option allows to set a maximum number of messages to receive before stopping.

See references/moat_mqtt_sub reference documentation for details about available options and settings.

URL Scheme

MoaT-MQTT command line tools use the `--url` to establish a network connection with the broker. The `--url` parameter value must conform to the MQTT URL scheme. The general accepted form is :

```
{mqtt,ws}[s]://[username][:password]@host.domain[:port]
```

Here are some examples of valid URLs:

```
mqtt://localhost
mqtt://localhost:1884
mqtt://user:password@localhost
ws://test.mosquitto.org
wss://user:password@localhost
```

Running a broker

`moat mqtt broker` is a command-line tool for running a MQTT broker:

```
$ moat mqtt broker
[2015-11-06 22:45:16,470] :: INFO - Listener 'default' bind to 0.0.0.0:1883 (max_
↪connections=-1)
```

See references/moat_mqtt_broker reference documentation for details about available options and settings.

4.2 Changelog

4.2.1 0.10

- Ported to anyio, thus works with asyncio+trio+curio.
- Refactored so that closed connections don't affect message delivery.

4.2.2 0.9.5

- fix more issues
- fix a few issues

4.2.3 0.9.2

- fix a few issues

4.2.4 0.9.1

- See commit log

4.2.5 0.9.0

- fix a serie of issues
- improve plugin performance
- support Python 3.6
- upgrade to `websockets` 3.3.0

4.2.6 0.8.0

- fix a serie of issues

4.2.7 0.7.3

- fix deliver message client method to raise TimeoutError (#40)
- fix topic filter matching in broker (#41)

Version 0.7.2 has been jumped due to troubles with pypi...

4.2.8 0.7.1

- Fix duplicated $SYS topic name .

4.2.9 0.7.0

- Fix a serie of issues reported by Christoph Krey

4.2.10 0.6.3

- Fix issue #22.

4.2.11 0.6.2

- Fix issue #20 (`mqtt` subprotocol was missing).
- Upgrade to `websockets` 3.0.

4.2.12 0.6.1

- Fix issue #19

4.2.13 0.6

- Added compatibility with Python 3.5.
- Rewritten documentation.
- Add command-line tools references/distmqtt, references/distmqtt_pub and references/distmqtt_sub.

4.3 References

Reference documentation for MoaT-MQTT console scripts and programming API.

4.3.1 Console scripts

- moat_mqtt_pub : MQTT client for publishing messages to a broker
- moat_mqtt_sub : MQTT client for subscribing to a topics and retrieved published messages
- moat_mqtt_broker : Autonomous MQTT broker

4.3.2 Programming API

- *MQTTClient API* : MQTT client API reference
- *Broker API reference* : MQTT broker API reference
- *Common API* : Common API

TBD

MQTTClient API

The `MQTTClient` class implements the client part of MQTT protocol. It can be used to publish and/or subscribe MQTT message on a broker accessible on the network through TCP or websocket protocol, both secured or unsecured.

Usage examples

Subscriber

The example below shows how to write a simple MQTT client which subscribes a topic and prints every messages received from the broker :

```
import logging
import anyio

from moat.mqtt.client import open_mqttclient, ClientException
from moat.mqtt.mqtt.constants import QOS_1, QOS_2

logger = logging.getLogger(__name__)

async def uptime_coro():
    async with open_mqttclient(uri='mqtt://test.mosquitto.org/') as C:
        # Subscribe to '$SYS/broker/uptime' with QOS=1
        # Subscribe to '$SYS/broker/load/#' with QOS=2
        await C.subscribe([
                ('$SYS/broker/uptime', QOS_1),
                ('$SYS/broker/load/#', QOS_2),
             ])
        for i in range(1, 100):
            message = await C.deliver_message()
            packet = message.publish_packet
            print("%d:  %s => %s" % (i, packet.variable_header.topic_name, str(packet.
↪payload.data)))
        await C.unsubscribe(['$SYS/broker/uptime', '$SYS/broker/load/#'])

if __name__ == '__main__':
    formatter = "[%(asctime)s] %(name)s {%(filename)s:%(lineno)d} %(levelname)s -
↪%(message)s"
    logging.basicConfig(level=logging.DEBUG, format=formatter)
    anyio.run(uptime_coro)
```

This code has a problem: there's one central dispatcher which needs to know all message types. Fortunately *moat.mqtt* has a built-in dispatcher.

```
async def show(C, topic, qos):
    async with C.subscription(topic, qos) as sub:
        count = 0
        async for message in sub:
            packet = message.publish_packet
            print("%d:  %s => %s" % (i, packet.variable_header.topic_name, str(packet.
↪payload.data)))
            count += 1
            if count >= 100:
            break

async def uptime_coro():
    async with open_mqttclient(uri='mqtt://test.mosquitto.org/') as C:
        # Subscribe to '$SYS/broker/uptime' with QOS=1
        # Subscribe to '$SYS/broker/load/#' with QOS=2
        async with anyio.create_task_group() as tg:
           tg.start_soon(show, C, '$SYS/broker/uptime', QOS_1);
           tg.start_soon(show, C, '$SYS/broker/load/#', QOS_2);

if __name__ == '__main__':
    formatter = "[%(asctime)s] %(name)s {%(filename)s:%(lineno)d} %(levelname)s -
↪%(message)s"
    logging.basicConfig(level=logging.DEBUG, format=formatter)
    anyio.run(uptime_coro)
```

Publisher

The example below uses the `MQTTClient` class to implement a publisher. This test publish 3 messages asynchronously to the broker on a test topic. For the purposes of the test, each message is published with a different Quality Of Service.

```
import logging
import anyio

from moat.mqtt.client import MQTTClient
from moat.mqtt.mqtt.constants import QOS_0, QOS_1, QOS_2

logger = logging.getLogger(__name__)

async def test_coro():
    """Publish in parallel"""
    async with open_mqttclient(uri='mqtt://test.mosquitto.org/') as C:
        async with anyio.create_task_group() as tg:
            tg.start_soon(C.publish,'a/b', b'TEST MESSAGE WITH QOS_0')
            tg.start_soon(C.publish,'a/b', b'TEST MESSAGE WITH QOS_1', qos=QOS_1)),
            tg.start_soon(C.publish,'a/b', b'TEST MESSAGE WITH QOS_2', qos=QOS_2)),
        logger.info("messages published")


async def test_coro2():
    """Publish sequentially"""
    try:
        async with open_mqttclient(uri='mqtt://test.mosquitto.org/') as C:
           await C.publish('a/b', b'TEST MESSAGE WITH QOS_0', qos=QOS_0)
           await C.publish('a/b', b'TEST MESSAGE WITH QOS_1', qos=QOS_1)
```

(continues on next page)

(continued from previous page)

```
            await C.publish('a/b', b'TEST MESSAGE WITH QOS_2', qos=QOS_2)
            logger.info("messages published")
        except ConnectException as ce:
            logger.error("Connection failed: %s", ce)


if __name__ == '__main__':
    formatter = "[%(asctime)s] %(name)s {%(filename)s:%(lineno)d} %(levelname)s -
↪%(message)s"
    logging.basicConfig(level=logging.DEBUG, format=formatter)
    anyio.run(test_coro)
    anyio.run(test_coro2)
```

Both coroutines have the same results except that `test_coro()` sends its messages in parallel, and thus is probably a bit faster.

Reference

MQTTClient API

MQTTClient configuration

Typically, you create a `MQTTClient` instance with an async context manager, i.e. by way of `async with open_mqttclient()()`. This context manager creates a taskgroup for the client's housekeeping tasks to run in.

`open_mqttclient()` accepts a `config` parameter which allows to setup some behaviour and defaults settings. This argument must be a Python dictionary which may contain the following entries:

- `keep_alive`: keep alive interval (in seconds) to send when connecting to the broker (defaults to `10` seconds). `MQTTClient` will *auto-ping* the broker if no message is sent within the keep-alive interval. This avoids disconnection from the broker.
- `ping_delay`: *auto-ping* delay before keep-alive times out (defaults to `1` seconds). This should be larger than twice the worst-case roundtrip between your client and the broker.
- `default_qos`: Default QoS (`0`) used by `publish()` if `qos` argument is not given.
- `default_retain`: Default retain (`False`) used by `publish()` if `retain` argument is not given.
- `auto_reconnect`: enable or disable auto-reconnect feature (defaults to `True`).
- `reconnect_max_interval`: maximum interval (in seconds) to wait before two connection retries (defaults to `10`).
- `reconnect_retries`: maximum number of connect retries (defaults to `2`). Negative value will cause client to reconnect infinietly.
- `codec`: the codec to use by default. May be overridden.
- `codec_params`: Config values to use with a particular codec. Indexed by codec name.

Default QoS and default retain can also be overriden by adding a `topics` entry with may contain QoS and retain values for specific topics. See the following example:

```
config = {
    'keep_alive': 10,
    'ping_delay': 1,
```

(continues on next page)

(continued from previous page)

```
    'default_qos': 0,
    'default_retain': False,
    'auto_reconnect': True,
    'reconnect_max_interval': 5,
    'reconnect_retries': 10,
    'codec': 'utf8',
    'codec_params': {
        'bool': {on='on',off='off'}, ## default, actually
        'BOOL': {on='ON',off='OFF',name='bool'}
        'yesno': {on='yes',off='no', name='bool'}
    },
    'topics': {
        '/test': { 'qos': 1 },
        '/some_topic': { 'qos': 2, 'retain': True }
    }
}
```

With this setting any message published will set with QOS_0 and retain flag unset except for

- messages sent to `/test` topic will be sent with QOS_1
- messages sent to `/some_topic` topic will be sent with QOS_2 and retained

Also, 'codec="yesno"' will only accept a `bool` as message, and translate that to "yes" and "no" messages.

In any case, any `qos` and `retain` arguments passed to method `publish()` will override these settings.

Broker API reference

The `Broker` class provides a complete MQTT 3.1.1 broker implementation. This class allows Python developers to embed a MQTT broker in their own applications.

Usage example

The following example shows how to start a broker using the default configuration:

```
import logging
import anyio
import os
from moat.mqtt.broker import open_broker


async def broker_coro():
    async with create_broker() as broker:
        while True:
            await anyio.sleep(99999)


if __name__ == '__main__':
    formatter = "[%(asctime)s] :: %(levelname)s :: %(name)s :: %(message)s"
    logging.basicConfig(level=logging.INFO, format=formatter)
    anyio.run(broker_coro)
```

When executed, this script runs the `broker_coro` until it completes. `broker_coro` creates a `Broker` instance. Once completed, the loop is ran forever, making this script never stop . . .

Reference

Broker API

Typically, you create a `Broker` instance by way of `async with` *`create_broker()`*(). This context manager creates a taskgroup for the client's housekeeping tasks to run in.

`moat.mqtt.broker.`**`create_broker`**`()`

If using an async context manager doesn't fit your code, you can pass your own taskgroup and explicitly start (and stop) the broker. However, the broker may leak some tasks, thus using *`create_broker()`* is strongly recommended.

Broker configuration

~moat.mqtt.broker.create_broker accepts a `config` parameter which allows to setup some behaviour and defaults settings. This argument must be a Python dictionary. For convenience, it is presented below as a YAML file[1].

```
listeners:
    default:
        max-connections: 50000
        type: tcp
    my-tcp-1:
        bind: 127.0.0.1:1883
    my-tcp-2:
        bind: 1.2.3.4:1884
        max-connections: 1000
    my-tcp-ssl-1:
        bind: 127.0.0.1:8885
        ssl: on
        cafile: /some/cafile
        capath: /some/folder
        capath: certificate data
        certfile: /some/certfile
        keyfile: /some/key
    my-ws-1:
        bind: 0.0.0.0:8080
        type: ws
timeout-disconnect-delay: 2
auth:
    plugins: ['auth.anonymous'] #List of plugins to activate for authentication among␣
↪all registered plugins
    allow-anonymous: true / false
    password-file: "/some/passwd_file"
topic-check:
    enabled: true / false  # Set to False if topic filtering is not needed
    plugins: ['topic_acl'] #List of plugins to activate for topic filtering among all␣
↪registered plugins
    acl:
        # username: [list of allowed topics]
        username1: ['repositories/+/master', 'calendar/#', 'data/memes']  # List of␣
↪topics on which client1 can publish and subscribe
        username2: ...
        anonymous: []  # List of topics on which an anonymous client can publish and␣
↪subscribe
```

[1] See PyYAML for loading YAML files as Python dict.

The `listeners` section allows to define network listeners which must be started by the `Broker`. Several listeners can be setup. `default` subsection defines common attributes for all listeners. Each listener can have the following settings:

- `bind`: IP address and port binding.
- `max-connections`: Set maximum number of active connection for the listener. `0` means no limit.
- `type`: transport protocol type; can be `tcp` for classic TCP listener or `ws` for MQTT over websocket.
- `ssl` enables (`on`) or disable secured connection over the transport protocol.
- `cafile`, `cadata`, `certfile` and `keyfile` : mandatory parameters for SSL secured connections.

The `auth` section setup authentication behaviour:

- `plugins`: defines the list of activated plugins. Note the plugins must be defined in the `moat.mqtt.broker.plugins` entry point.
- `allow-anonymous` : used by the internal `moat.mqtt.plugins.authentication.AnonymousAuthPlugin` plugin. This parameter enables (`on`) or disable anonymous connection, ie. connection without username.
- `password-file` : used by the internal `moat.mqtt.plugins.authentication.FileAuthPlugin` plugin. This parameter gives to path of the password file to load for authenticating users.

The `topic-check` section setup access control policies for publishing and subscribing to topics:

- `enabled`: set to true if you want to impose an access control policy. Otherwise, set it to false.
- `plugins`: defines the list of activated plugins. Note the plugins must be defined in the `moat.mqtt.broker.plugins` entry point.
- **additional parameters: depending on the plugin used for access control, additional parameters should be added.**
  - **In case of `topic_acl` plugin, the Access Control List (ACL) must be defined in the parameter `acl`.**
    - For each username, a list with the allowed topics must be defined.
    - If the client logs in anonymously, the `anonymous` entry within the ACL is used in order to grant/deny subscriptions.

Common API

This document describes `Moat-MQTT` common API both used by *MQTTClient API* and *Broker API reference*.

Reference

ApplicationMessage

4.4 License

```
The MIT License (MIT)

Copyright (c) 2015 Nicolas JOUANIN
```

(continues on next page)

(continued from previous page)

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

Index

M