aMQTT Documentation

Release 0.10.0-alpha.1

Nicolas Jouanin

Feb 09, 2023

CONTENTS

aMQTT is an open source MQTT client and broker implementation.

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

CHAPTER

ONE

FEATURES

aMQTT 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

TWO

REQUIREMENTS

aMQTT is built on Python's *asyncio*. It requires Python 3.6 or newer.

CHAPTER

THREE

INSTALLATION

It is not recommended to install third-party library in Python system packages directory. The preferred way for installing aMQTT 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, aMQTT can be installed with the following command

```
(venv) $ pip install amqtt
```

`pip` will download and install aMQTT and all its dependencies.

CHAPTER

FOUR

USER GUIDE

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

If you want to develop an application which needs to connect to a MQTT broker, the *MQTTClient API* documentation explains how to use aMQTT 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 `aMQTT` 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 a autonomous MQTT broker

These scripts are installed automatically when installing `aMQTT` with the following command

```
(venv) $ pip install amqtt
```

4.1.1 Publishing messages

`amqtt_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 as `/test` topic on is as simple as :

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

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

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

amqtt_pub can read from file or stdin and use data read as message payload:

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

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

4.1.2 Subscribing a topic

amqtt_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 :

```
$ amqtt_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/amqtt_sub reference documentation for details about available options and settings.

4.1.3 URL Scheme

aMQTT 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 URL:

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

4.1.4 Running a broker

amqtt is a command-line tool for running a MQTT broker:

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

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

4.2 Changelog

4.3 0.11.0 unreleased

- removed hbmqtt compatibility, importing amqtt is now mandatory
- removed loop parameter from all functions
- Python 3.7 to 3.10 compatibility
- Fixed a major bug in plugin api, see https://github.com/Yakifo/amqtt/pull/92
- Fixed a major bug due to CancelledError semantics changed in Python 3.8, see https://github.com/Yakifo/amqtt/pull/134

4.4 0.10.0 - 2021-08-04

- first release under new package name: amqtt
- reworked unit tests
- dropped support for python3.5 and earlier
- added support for python3.8 and 3.9
- Pass in loop to PluginManager, from https://github.com/beerfactory/hbmqtt/pull/126
- Fixes taboo topic checking without session username, from https://github.com/beerfactory/hbmqtt/pull/151
- Move scripts module into hbmqtt module, from https://github.com/beerfactory/hbmqtt/pull/167
- Download mosquitto certificate on the fly
- importing *hbmqtt* is deprecated, use *amqtt*
- Security fix: If an attacker could produce a KeyError inside an authentication plugin, the authentication was accepted instead of rejected

4.4.1 0.9.5

- fix more issues
- fix a few issues

4.4.2 0.9.2

- fix a few issues

4.4.3 0.9.1

- See commit log

4.4.4 0.9.0

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

4.4.5 0.8.0

- fix a serie of issues

4.4.6 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…

4.4.7 0.7.1

- Fix duplicated $SYS topic name .

4.4.8 0.7.0

- Fix a serie of issues reported by Christoph Krey

4.4.9 0.6.3

- Fix issue #22.

4.4.10 0.6.2

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

4.4.11 0.6.1

- Fix issue #19

4.4.12 0.6

- Added compatibility with Python 3.5.
- Rewritten documentation.
- Add command-line tools references/hbmqtt, references/hbmqtt_pub and references/hbmqtt_sub.

4.5 Transitioning from HBMQTT to aMQTT

This document is for those porting from HBMQTT to aMQTT. Basically you can search and replace `hbmqtt` with `amqtt` and all should work out. Details below.

4.5.1 Imports

The module changed from `hbmqtt` to `amqtt`. For the 0.10.x releases it will still be possible to `import hbmqtt`. In 0.11.x only `amqtt` will work.

Since the `amqtt` package does provide a `hbmqtt` module, installing the `hbmqtt` package in the same python installation is not possible.

4.5.2 Random Client ID

When not providing a client_id, a random id is automatically generated. These names were also changed from `hbmqtt/<random>` to `amqtt/<random>`.

4.5.3 plugins / entrypoints

If you make use of python's entrypoint system to build aMQTT plugins, make sure to use the `amqtt.*.plugins` names instead of `hbmqtt.*.plugins` names. During the transition plugins with `hbmqtt` entrypoint should keep working for 0.10.x releases.

4.5.4 CLI tools

Will also be renamed.

4.6 References

Reference documentation for aMQTT console scripts and programming API.

4.6.1 Console scripts

- amqtt_pub : MQTT client for publishing messages to a broker
- amqtt_sub : MQTT client for subscribing to a topics and retrieved published messages
- amqtt : Autonomous MQTT broker

4.6.2 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 asyncio

from amqtt.client import MQTTClient, ClientException
from amqtt.mqtt.constants import QOS_1, QOS_2

logger = logging.getLogger(__name__)

async def uptime_coro():
    C = MQTTClient()
    await C.connect('mqtt://test.mosquitto.org/')
    # 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),
        ])
```

(continues on next page)

(continued from previous page)

```
    try:
        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/#'])
        await C.disconnect()
    except ClientException as ce:
        logger.error("Client exception: %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)
    asyncio.get_event_loop().run_until_complete(uptime_coro())
```

When executed, this script gets the default event loop and asks it to run the `uptime_coro` until it completes. `uptime_coro` starts by initializing a `MQTTClient` instance. The coroutine then call `connect()` to connect to the broker, here `test.mosquitto.org`. Once connected, the coroutine subscribes to some topics, and then wait for 100 messages. Each message received is simply written to output. Finally, the coroutine unsubscribes from topics and disconnects from the broker.

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. This example also shows to method for publishing message asynchronously.

```
import logging
import asyncio

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

logger = logging.getLogger(__name__)

async def test_coro():
    C = MQTTClient()
    await C.connect('mqtt://test.mosquitto.org/')
    tasks = [
        asyncio.ensure_future(C.publish('a/b', b'TEST MESSAGE WITH QOS_0')),
        asyncio.ensure_future(C.publish('a/b', b'TEST MESSAGE WITH QOS_1', qos=QOS_1)),
        asyncio.ensure_future(C.publish('a/b', b'TEST MESSAGE WITH QOS_2', qos=QOS_2)),
    ]
    await asyncio.wait(tasks)
    logger.info("messages published")
    await C.disconnect()


async def test_coro2():
```

(continues on next page)

(continued from previous page)

```
    try:
        C = MQTTClient()
        ret = await C.connect('mqtt://test.mosquitto.org:1883/')
        message = await C.publish('a/b', b'TEST MESSAGE WITH QOS_0', qos=QOS_0)
        message = await C.publish('a/b', b'TEST MESSAGE WITH QOS_1', qos=QOS_1)
        message = await C.publish('a/b', b'TEST MESSAGE WITH QOS_2', qos=QOS_2)
        #print(message)
        logger.info("messages published")
        await C.disconnect()
    except ConnectException as ce:
        logger.error("Connection failed: %s" % ce)
        asyncio.get_event_loop().stop()


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

As usual, the script runs the publish code through the async loop. `test_coro()` and `test_coro()` are ran in sequence. Both do the same job. `test_coro()` publish 3 messages in sequence. `test_coro2()` publishes the same message asynchronously. The difference appears the looking at the sequence of MQTT messages sent.

`test_coro()` achieves:

```
amqtt/YDYY;NNRpYQSy3?o -out-> PublishPacket(ts=2015-11-11 21:54:48.843901,␣
↪fixed=MQTTFixedHeader(length=28, flags=0x0), variable=PublishVariableHeader(topic=a/b,␣
↪packet_id=None), payload=PublishPayload(data="b'TEST MESSAGE WITH QOS_0'"))
amqtt/YDYY;NNRpYQSy3?o -out-> PublishPacket(ts=2015-11-11 21:54:48.844152,␣
↪fixed=MQTTFixedHeader(length=30, flags=0x2), variable=PublishVariableHeader(topic=a/b,␣
↪packet_id=1), payload=PublishPayload(data="b'TEST MESSAGE WITH QOS_1'"))
amqtt/YDYY;NNRpYQSy3?o <-in-- PubackPacket(ts=2015-11-11 21:54:48.979665,␣
↪fixed=MQTTFixedHeader(length=2, flags=0x0), variable=PacketIdVariableHeader(packet_
↪id=1), payload=None)
amqtt/YDYY;NNRpYQSy3?o -out-> PublishPacket(ts=2015-11-11 21:54:48.980886,␣
↪fixed=MQTTFixedHeader(length=30, flags=0x4), variable=PublishVariableHeader(topic=a/b,␣
↪packet_id=2), payload=PublishPayload(data="b'TEST MESSAGE WITH QOS_2'"))
amqtt/YDYY;NNRpYQSy3?o <-in-- PubrecPacket(ts=2015-11-11 21:54:49.029691,␣
↪fixed=MQTTFixedHeader(length=2, flags=0x0), variable=PacketIdVariableHeader(packet_
↪id=2), payload=None)
amqtt/YDYY;NNRpYQSy3?o -out-> PubrelPacket(ts=2015-11-11 21:54:49.030823,␣
↪fixed=MQTTFixedHeader(length=2, flags=0x2), variable=PacketIdVariableHeader(packet_
↪id=2), payload=None)
amqtt/YDYY;NNRpYQSy3?o <-in-- PubcompPacket(ts=2015-11-11 21:54:49.092514,␣
↪fixed=MQTTFixedHeader(length=2, flags=0x0), variable=PacketIdVariableHeader(packet_
↪id=2), payload=None)fixed=MQTTFixedHeader(length=2, flags=0x0),␣
↪variable=PacketIdVariableHeader(packet_id=2), payload=None)
```

while `test_coro2()` runs:

```
amqtt/LYRf52W[56SOjW04 -out-> PublishPacket(ts=2015-11-11 21:54:48.466123,␣
```

(continues on next page)

(continued from previous page)

```
↪fixed=MQTTFixedHeader(length=28, flags=0x0), variable=PublishVariableHeader(topic=a/b,␣
↪packet_id=None), payload=PublishPayload(data="b'TEST MESSAGE WITH QOS_0'"))
amqtt/LYRf52W[56SOjW04 -out-> PublishPacket(ts=2015-11-11 21:54:48.466432,␣
↪fixed=MQTTFixedHeader(length=30, flags=0x2), variable=PublishVariableHeader(topic=a/b,␣
↪packet_id=1), payload=PublishPayload(data="b'TEST MESSAGE WITH QOS_1'"))
amqtt/LYRf52W[56SOjW04 -out-> PublishPacket(ts=2015-11-11 21:54:48.466695,␣
↪fixed=MQTTFixedHeader(length=30, flags=0x4), variable=PublishVariableHeader(topic=a/b,␣
↪packet_id=2), payload=PublishPayload(data="b'TEST MESSAGE WITH QOS_2'"))
amqtt/LYRf52W[56SOjW04 <-in-- PubackPacket(ts=2015-11-11 21:54:48.613062,␣
↪fixed=MQTTFixedHeader(length=2, flags=0x0), variable=PacketIdVariableHeader(packet_
↪id=1), payload=None)
amqtt/LYRf52W[56SOjW04 <-in-- PubrecPacket(ts=2015-11-11 21:54:48.661073,␣
↪fixed=MQTTFixedHeader(length=2, flags=0x0), variable=PacketIdVariableHeader(packet_
↪id=2), payload=None)
amqtt/LYRf52W[56SOjW04 -out-> PubrelPacket(ts=2015-11-11 21:54:48.661925,␣
↪fixed=MQTTFixedHeader(length=2, flags=0x2), variable=PacketIdVariableHeader(packet_
↪id=2), payload=None)
amqtt/LYRf52W[56SOjW04 <-in-- PubcompPacket(ts=2015-11-11 21:54:48.713107,␣
↪fixed=MQTTFixedHeader(length=2, flags=0x0), variable=PacketIdVariableHeader(packet_
↪id=2), payload=None)
```

Both coroutines have the same results except that `test_coro2()` manages messages flow in parallel which may be more efficient.

Reference

MQTTClient API

MQTTClient configuration

The `MQTTClient __init__` method accepts a `config` parameter which allow to setup some behaviour and defaults settings. This argument must be a Python dict object which may contain the following entries:

- `keep_alive`: keep alive (in seconds) to send when connecting to the broker (defaults to `10` seconds). `MQTTClient` will *auto-ping* the broker if not 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).
- `default_qos`: Default QoS (`0`) used by `publish()` if `qos` argument is not given.
- `default_retain`: Default retain (`False`) used by `publish()` if `qos` 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.

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

```
config = {
    'keep_alive': 10,
    'ping_delay': 1,
    'default_qos': 0,
    'default_retain': False,
    'auto_reconnect': True,
    'reconnect_max_interval': 5,
    'reconnect_retries': 10,
    '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 : they will be sent with QOS_1
- messages sent to `some_topic` topic : they will be sent with QOS_2 and retain flag set

In any case, the `qos` and `retain` argument values 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 asyncio
import os
from amqtt.broker import Broker


async def broker_coro():
    broker = Broker()
    await broker.start()


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

When executed, this script gets the default event loop and asks it to run the `broker_coro` until it completes. `broker_coro` creates `Broker` instance and then `start()` the broker for serving. Once completed, the loop is ran forever, making this script never stop …

Reference

Broker API

Broker configuration

The `Broker __init__` method accepts a `config` parameter which allow to setup some behaviour and defaults settings. This argument must be a Python dict object. For convinience, 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
```

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.

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

- `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 `amqtt.broker.plugins` entry point.
- `allow-anonymous` : used by the internal `amqtt.plugins.authentication.AnonymousAuthPlugin` plugin. This parameter enables (`on`) or disable anonymous connection, ie. connection without username.
- `password-file` : used by the internal `amqtt.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 `amqtt.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 `aMQTT` common API both used by *MQTTClient API* and *Broker API reference*.

Reference

ApplicationMessage

4.7 License

```
The MIT License (MIT)

Copyright (c) 2015 Nicolas JOUANIN
Copyright (c) 2021 aMQTT Contributers (https://github.com/Yakifo/amqtt/graphs/
↪contributors)

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:
```

(continues on next page)

(continued from previous page)

```
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.
```