mosquittoChat Documentation

Release 1.1.0

Anirban Roy Das

Jun 12, 2017

Contents

An MQTT protocol based Chat-Server/Chat-System using Mosquitto Broker, tornado as web server, sockjs in client(browser) side javascript library, sockjs-tornado as sockjs implementation on server side and paho-mqtt (mqtt python client).

Home Page : https://pypi.python.org/pypi/mosquittoChat

CHAPTER 1

Details

Author Anirban Roy Das

Email anirban.nick@gmail.com

CHAPTER 2

Documentation:

Overview

mosquittochat is an MQTT protocol based simple Chat Server which can be set up locally to chat in your LAN. It supports both **Public Chat** among all participants connected simultaneously at a particular time and also **Private Chat** betweent those individual participants.

It uses the MQTT protocol to implement the real time message passing system. **MQTT** is implemented in many languages and in many softwares, one of such is Mosquitto , which is a message broker implementing the MQTT protocol.

The connection is created using the sockjs protocol. **SockJS** is implemented in many languages, primarily in Javascript to talk to the servers in real time, which tries to create a duplex bi-directional connection between the **Client(browser)** and the **Server**. Ther server should also implement the **sockjs** protocol. Thus using the sockjs-tornado library which exposes the **sockjs** protocol in Tornado server.

It first tries to create a Websocket connection, and if it fails then it fallbacks to other transport mechanisms, such as **Ajax**, **long polling**, etc. After the connection is established, the tornado server**(sockjs-tornado)** connects to **Mosquitto** via MQTT protocol using the **MQTT Python Client Library**, paho-mqtt.

Thus the connection is *web-browser* to *tornado* to *mosquitto* and vice versa.

Features

Technical Specs

sockjs-client Advanced Websocket Javascript Client

Tornado Async Python Web Library + Web Server

sockjs-tornado SockJS websocket server implementation for Tornado

MQTT Machine-to-Machine (M2M)/"Internet of Things" connectivity protocol

paho-mqtt	MQTT Python Client Library
Mosquitto	A Message Broker implementing MQTT in C
pytest	Python testing library and test runner with awesome test discobery
pytest-flask	Pytest plugin for flask apps, to test fask apps using pytest library.
Uber's Test-Double	Test Double library for python, a good alternative to the mock library
Jenkins (Optional)	A Self-hosted CI server
Travis-CI (Optional)	A hosted CI server free for open-source projecs
Docker	A containerization tool for better devops

Feature Specs

- Public chat
- Shows who joined and who left
- Shows list of users online/offline
- Show last seen of offline features
- Shows who is typing and who is not - typing indicator
- Shows number of people online in public chat
- Join/Leave chat room features
- Microservice
- Testing using Docker and Docker Compose
- CI servers like Jenkins, Travis-CI

Installation

There are two types of Installation. One using mosquittoChat as a binary by installaing from pip and running the application in the local machine directly. Another method is running the application from Docker. Hence another set of installation steps for the Docker use case.

[Docker Method] Prerequisite (Optional)

To safegurad secret and confidential data leakage via your git commits to public github repo, check `git-secrets`.

This git secrets project helps in preventing secrete leakage by mistake.

[Docker Method] Dependencies

1. Docker
2. Make (Makefile)

See, there are so many technologies used mentioned in the tech specs and yet the dependencies are just two. This is the power of Docker.

[Docker Method] Install

- **Step 1 - Install Docker**

  Follow my another github project, where everything related to DevOps and scripts are mentioned along with setting up a development environemt to use Docker is mentioned.

  - Project: https://github.com/anirbanroydas/DevOps
  - Go to setup directory and follow the setup instructions for your own platform, linux/macos

- **Step 2 - Install Make**

```
# (Mac Os)
$ brew install automake

# (Ubuntu)
$ sudo apt-get update
$ sudo apt-get install make
```

- **Step 3 - Install Dependencies**

  Install the following dependencies on your local development machine which will be used in various scripts.

  1. openssl
  2. ssh-keygen
  3. openssh

[Standalone Binary Method] Prerequisites

1. python 2.7+
2. tornado
3. sockjs-tornado
4. sockjs-client
5. paho-mqtt
6. mosquitto

[Standalone Binary Method] Install

```
$ pip install mosquittoChat
```

If above dependencies do not get installed by the above command, then use the below steps to install them one by one.

Step 1 - Install pip

Follow the below methods for installing pip. One of them may help you to install pip in your system.

- **Method 1 -** https://pip.pypa.io/en/stable/installing/
- **Method 2 -** http://ask.xmodulo.com/install-pip-linux.html
- **Method 3 -** If you installed python on MAC OS X via `brew install python`, then **pip** is already installed along with python.

Step 2 - Install tornado

```
$ pip install tornado
```

Step 3 - Install sockjs-tornado

```
$ pip install sockjs-tornado
```

Step 4 - Install paho-mqtt

```
$ pip install paho-mqtt
```

Step 5 - Install Mosquitto

- *For* `Mac` *Users*

  1. Brew Install Mosquitto

  ```
  $ brew install mosquitto
  ```

  2. Configure mosquitto, by modifying the file at `/usr/local/etc/mosquitto/mosquitto.conf`.

- *For* `Ubuntu/Linux` *Users*

  1. Enable mosquitto repository (optional)

     First Try directly, if it doesn't work, then follow this step and continue after this.:

     ```
     $ sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
     ```

  2. Update the sources with our new addition from above

  ```
  $ apt-get update
  ```

  3. And finally, download and install Mosquitto

  ```
  $ sudo apt-get install mosquitto
  ```

  4. Configure mosquitto, by modifying the file at `/usr/local/etc/mosquitto/mosquitto.conf`.

CI Setup

If you are using the project in a CI setup (like travis, jenkins), then, on every push to github, you can set up your travis build or jenkins pipeline. Travis will use the `.travis.yml` file and Jenknis will use the `Jenkinsfile` to do their jobs. Now, in case you are using Travis, then run the Travis specific setup commands and for Jenkins run the Jenkins specific setup commands first. You can also use both to compare between there performance.

The setup keys read the values from a `.env` file which has all the environment variables exported. But you will notice an example `env` file and not a `.env` file. Make sure to copy the `env` file to `.env` and **change/modify** the actual variables with your real values.

The `.env` files are not commited to git since they are mentioned in the `.gitignore` file to prevent any leakage of confidential data.

After you run the setup commands, you will be presented with a number of secure keys. Copy those to your config files before proceeding.

NOTE: This is a one time setup. **NOTE:** Check the setup scripts inside the `scripts/` directory to understand what are the environment variables whose encrypted keys are provided. **NOTE:** Don't forget to **Copy** the secure keys to your `.travis.yml` or `Jenkinsfile`

NOTE: If you don't want to do the copy of `env` to `.env` file and change the variable values in `.env` with your real values then you can just edit the `travis-setup.sh` or `jenknis-setup.sh` script and update the values their directly. The scripts are in the `scripts/` project level directory.

IMPORTANT: You have to run the `travis-setup.sh` script or the `jenkins-setup.sh` script in your local machine before deploying to remote server.

Travis Setup

These steps will encrypt your environment variables to secure your confidential data like api keys, docker based keys, deploy specific keys.

```
$ make travis-setup
```

Jenkins Setup

These steps will encrypt your environment variables to secure your confidential data like api keys, docker based keys, deploy specific keys.

```
$ make jenkins-setup
```

Usage

There are two types of Usage. One using mosquittoChat as a binary by installaing from pip and running the application in the local machine directly. Another method is running the application from Docker. Hence another set of usage steps for the Docker use case.

[Docker Method]

After having installed the above dependencies, and ran the **Optional** (If not using any CI Server) or **Required** (If using any CI Server) **CI Setup** Step, then just run the following commands to use it:

You can run and test the app in your local development machine or you can run and test directly in a remote machine. You can also run and test in a production environment.

[Docker Method] Run

The below commands will start everythin in development environment. To start in a production environment, suffix `-prod` to every **make** command.

For example, if the normal command is `make start`, then for production environment, use `make start-prod`. Do this modification to each command you want to run in production environment.

Exceptions: You cannot use the above method for test commands, test commands are same for every environment. Also the `make system-prune` command is standalone with no production specific variation (Remains same in all environments).

- **Start Applcation**

```
$ make clean
$ make build
$ make start

# OR

$ docker-compose up -d
```

- **Stop Application**

```
$ make stop

# OR

$ docker-compose stop
```

- **Remove and Clean Application**

```
$ make clean

# OR

$ docker-compose rm --force -v
$ echo "y" | docker system prune
```

- **Clean System**

```
$ make system-prune

# OR

$ echo "y" | docker system prune
```

[Docker Method] Logging

- To check the whole application Logs

```
$ make check-logs

# OR

$ docker-compose logs --follow --tail=10
```

- To check just the python app's logs

```
$ make check-logs-app

# OR

$ docker-compose logs --follow --tail=10 identidock
```

[Standalone Binary Method] Run

After having installed mosquittoChat, just run the following commands to use it:

Mosquitto Server

1. *For* `Mac` *Users*

```
# start normally
$ mosquitto -c /usr/local/etc/mosquitto/mosquitto.conf

# If you want to run in background
$ mosquitto -c /usr/local/etc/mosquitto/mosquitto.conf -d

# start using brew services (doesn't work with tmux, athough there is a fix,␣
↪mentioned in one of the pull requests and issues)
$ brew services start mosquitto
```

2. *For* `Ubuntu/LInux` *Users*

```
# start normally
$ mosquitto -c /usr/local/etc/mosquitto/mosquitto.conf

# If you want to run in background
$ mosquitto -c /usr/local/etc/mosquitto/mosquitto.conf -d

# To start using service
$ sudo service mosquitto start

# To stop using service
$ sudo service mosquitto stop

# To restart using service
$ sudo service mosquitto restart

# To check the status
$ service mosquitto status
```

mosquittoChat Application

1. Start Server

```
$ mosquittoChat [options]
```

2. Options

   –port Port number where the chat server will start

   - **Example**

   ```
   $ mosquittoChat --port=9191
   ```

3. Stop mosquittoChat Server

   Click `Ctrl+C` to stop the server.

API

This contains all the modules and classes used to make the app. You can go through each of them for better understanding of the project.

Main View

This is the main view module which manages main tornado connections. This module provides request handlers for managing simple HTTP requests as well as Websocket requests.

Although the websocket requests are actually sockJs requests which follows the sockjs protcol, thus it provide interface to sockjs connection handlers behind the scene.

IndexHandler

class `mosquittoChat.apps.main.views.`**`IndexHandler`**(*application*, *request*, ***kwargs*)
This handler is a basic regular HTTP handler to serve the chatroom page.

`get`()
This method is called when a client does a simple GET request, all other HTTP requests like POST, PUT, DELETE, etc are ignored.

Returns Returns the rendered main requested page, in this case its the chat page, index.html

`__module__` **= 'mosquittoChat.apps.main.views'**

ChatWebsocketHandler

class `mosquittoChat.apps.main.views.`**`ChatWebsocketHandler`**(*session*)
Websocket Handler implementing the sockjs Connection Class which will handle the websocket/sockjs connections.

`on_open`(*info*)
This method is called when a websocket/sockjs connection is opened for the first time.

Parameters

- **`self`** – The object
- **`info`** – The information

Returns It returns the websocket object

`on_message`(*message*)
This method is called when a message is received via the websocket/sockjs connection created initially.

Parameters

- **`message`** (*`json string`*) – The message received via the connection.

- **`self`** – The object

`on_close`()
This method is called when a websocket/sockjs connection is closed.

Parameters **`self`** – The object

Returns Doesn't return anything, except a confirmation of closed connection back to web app.

`__module__` **= 'mosquittoChat.apps.main.views'**

PubSub Module

The pubsub module provides interface for the mosquitto client.

It provides classes to create mqtt clients vai paho-mqtt library to connect to mosquitto broker server, interact with and publish/subscribe to mosquitto via creating topics, methods to publish, subscribe/consume, stop consuming, start publishing, start connection, stop connection, acknowledge delivery by publisher, acknowledge receiving of messages by consumers and also add callbacks for various other events.

MosquittoClient

class `mosquittoChat.apps.mosquitto.pubsub.`**`MosquittoClient`**(*participants=1, name='user', clientid=None, clean_session=True, userdata=None, host='localhost', port=1883, keepalive=60, bind_address='', username='guest', password='guest'*)

This is a Mosquitto Client class that will create an interface to connect to mosquitto by creating mqtt clients.

It provides methods for connecting, diconnecting, publishing, subscribing, unsubscribing and also callbacks related to many different events like on_connect, on_message, on_publish, on_subscribe, on_unsubcribe, on_disconnect.

`__init__`(*participants=1, name='user', clientid=None, clean_session=True, userdata=None, host='localhost', port=1883, keepalive=60, bind_address='', username='guest', password='guest'*)

Create a new instance of the MosquittoClient class, passing in the client informaation, host, port, keepalive parameters.

Parameters

- **`participants`** (*`int`*) – number of participants available presently
- **`name`** (*`string`*) – name of client trying to connect to msoquitto
- **`clientid`** (*`string`*) – unique client id for a client-broker connection
- **`clean_session`** (*`bool`*) – whether to keep persistant connecion or not
- **`userdata`** (*`user defined data (can be int, string, or any object)`*) – user defined data of any type that is passed as the userdata parameter to callbacks. It may be updated at a later point with the user_data_set() function.
- **`host`** (*`string`*) – the hostname or IP address of the remote broker
- **`port`** (*`int`*) – the network port of the server host to connect to. Defaults to 1883. Note that the default port for MQTT over SSL/TLS is 8883 so if you are using tls_set() the port may need providing manually
- **`keepalive`** (*`int`*) – maximum period in seconds allowed between communications with the broker. If no other messages are being exchanged, this controls the rate at which the client will send ping messages to the broker
- **`bind_address`** (*`string`*) – the IP address of a local network interface to bind this client to, assuming multiple interfaces exist
- **`username`** (*`string`*) – username for authentication

- **password** (*string*) – password for authentication

_genid()

Method that generates unique clientids by calling base64.urlsafe_b64encode(os.urandom(32)).replace('=', 'e').

Returns Returns a unique urlsafe id

Return type string

start()

Method to start the mosquitto client by initiating a connection to mosquitto broker by using the connect method and staring the network loop.

setup_connection()

Method to setup the extra options like username,password, will set, tls_set etc before starting the connection.

create_client()

Method to create the paho-mqtt Client object which will be used to connect to mosquitto.

Returns Returns a mosquitto mqtt client object

Return type paho.mqtt.client.Client

setup_callbacks()

Method to setup all callbacks related to the connection, like on_connect, on_disconnect, on_publish, on_subscribe, on_unsubcribe etc.

connect()

This method connects to Mosquitto via returning the connection return code.

When the connection is established, the on_connect callback will be invoked by paho-mqtt.

Returns Returns a mosquitto mqtt connection return code, success, failure, error, etc

Return type int

on_connect(*client*, *userdata*, *flags*, *rc*)

This is a Callback method and is called when the broker responds to our connection request.

Parameters

- **client** – the client instance for this callback
- **userdata** – the private user data as set in Client() or userdata_set()
- **flags** (*dict*) – response flags sent by the broker
- **rc** (*int*) – the connection result

flags is a dict that contains response flags from the broker:

flags['session present'] - this flag is useful for clients that are using clean session set to 0 only. If a client with clean session=0, that reconnects to a broker that it has previously connected to, this flag indicates whether the broker still has the session information for the client. If 1, the session still exists.

The value of rc indicates success or not:

0: Connection successful 1: Connection refused - incorrect protocol version 2: Connection refused - invalid client identifier 3: Connection refused - server unavailable 4: Connection refused - bad username or password 5: Connection refused - not authorised 6-255: Currently unused.

start_ioloop()

Method to start ioloop for paho-mqtt mosquitto clients so that it can process read/write events for the sockets.

Using tornado's ioloop, since if we use any of the loop*() function provided by phao-mqtt library, it will either block the entire tornado thread, or it will keep on creating separate thread for each client if we use loop_start() fucntion.

We don't want to block thread or to create so many threads unnecessarily given python GIL.

Since the separate threads calls the loop() function indefinitely, and since its doing network io, its possible it may release GIL, but I haven't checked that yet, if that is the case, we can very well use loop_start().Pattern

But for now we will add handlers to tornado's ioloop().

`stop_ioloop`()

Method to stop ioloop for paho-mqtt mosquitto clients so that it cannot process any more read/write events for the sockets.

Actually the paho-mqtt mosquitto socket has been closed, so bascially this method removed the tornaod ioloop handler for this socket.

`_events_handler`(*fd*, *events*)

Handle IO/Event loop events, processing them.

Parameters

- **`fd`** (*`int`*) – The file descriptor for the events
- **`events`** (*`int`*) – Events from the IO/Event loop

`start_schedular`()

This method calls Tornado's PeriodicCallback to schedule a callback every few seconds, which calls paho mqtt client's loop_misc() function which keeps the connection open by checking for keepalive value and by keep sending pingreq and pingresp to moqsuitto broker.

`stop_schedular`()

This method calls stops the tornado's periodicCallback Schedular loop.

`disconnect`()

Method to disconnect the mqqt connection with mosquitto broker.

on_disconnect callback is called as a result of this method call.

`on_disconnect`(*client*, *userdata*, *rc*)

This is a Callback method and is called when the client disconnects from the broker.

`subscribe`(*topic_list=None*)

This method sets up the mqtt client to start subscribing to topics by accepting a list of tuples of topic and qos pairs.

The on_subscribe method is called as a callback if subscribing is succesfull or if it unsuccessfull, the broker returng the suback frame.

:param :topic_list: a tuple of (topic, qos), or, a list of tuple of format (topic, qos). :type :topic_list: list or tuple

`on_subscribe`(*client*, *userdata*, *mid*, *granted_qos*)

This is a Callback method and is called when the broker responds to a subscribe request.

The mid variable matches the mid variable returned from the corresponding subscribe() call. The granted_qos variable is a list of integers that give the QoS level the broker has granted for each of the different subscription requests.

Parameters

- **`client`** – the client which subscribed which triggered this callback
- **`userdata`** – the userdata associated with the client during its creation

- **`mid`** (*`int`*) – the message id value returned by the broker
- **`granted_qos`** (*`list`*) – list of integers that give the QoS level the broker has granted for each of the different subscription requests

`addNewMqttMosquittoClient`()
Method called after new mqtt connection is established and the client has started subsribing to atleast some topics, called by on_subscribe callback.

`sendMsgToWebsocket`(*msg*)
Method to send message to associated websocket.

Parameters **`msg`** (*`string, unicode or json encoded string or a dict`*) – the message to be sent to the websocket

`unsubscribe`(*topic_list=None*)
This method sets up the mqtt client to unsubscribe to topics by accepting topics as string or list.

The on_unsubscribe method is called as a callback if unsubscribing is succesfull or if it unsuccessfull.

Parameters **`topic_list`** (*`list of strings(topics)`*) – The topics to be unsubscribed from

`on_unsubscribe`(*client*, *userdata*, *mid*)
This is a Callback method and is called when the broker responds to an unsubscribe request. The mid variable matches the mid variable returned from t he corresponding unsubscribe() call.

Parameters

- **`client`** – the client which initiated unsubscribed
- **`userdata`** – the userdata associated with the client
- **`mid`** (*`int`*) – the message id value sent by the broker of the unsubscribe call.

`publish`(*topic*, *msg=None*, *qos=2*, *retain=False*)
If the class is not stopping, publish a message to MosquittoClient.

on_publish callback is called after broker confirms the published message.

Parameters

- **`topic`** (*`string`*) – The topic the message is to published to
- **`msg`** (*`string`*) – Message to be published to broker
- **`qos`** (*`int (0, 1 or 2)`*) – the qos of publishing message
- **`retain`** (*`bool`*) – Should the message be retained or not

`on_publish`(*client*, *userdata*, *mid*)
This is a Callback method and is called when a message that was to be sent using the publish() call has completed transmission to the broker. For messages with QoS levels 1 and 2, this means that the appropriate handshakes have completed.

For QoS 0, this simply means that the message has left the client. The mid variable matches the mid variable returned from the corresponding publish() call, to allow outgoing messages to be tracked.

This callback is important because even if the publish() call returns success, it does not always mean that the message has been sent.

Parameters

- **`client`** – the client who initiated the publish method
- **`userdata`** – the userdata associated with the client during its creation

- **mid** (*int*) – the message id sent by the broker

on_private_message (*client*, *userdata*, *msg*)

This is a Callback method and is called when a message has been received on a topic [private/cientid/msgs] that the client subscribes to.

Parameters

- **client** – the client who initiated the publish method
- **userdata** – the userdata associated with the client during its creation
- **msg** – the message sent by the broker

on_private_status (*client*, *userdata*, *msg*)

This is a Callback method and is called when a message has been received on a topic [private/cientid/status] that the client subscribes to.

Parameters

- **client** – the client who initiated the publish method
- **userdata** – the userdata associated with the client during its creation
- **msg** – the message sent by the broker

on_public_message (*client*, *userdata*, *msg*)

This is a Callback method and is called when a message has been received on a topic [public/msgs] that the client subscribes to.

Parameters

- **client** – the client who initiated the publish method
- **userdata** – the userdata associated with the client during its creation
- **msg** – the message sent by the broker

send_offline_status ()

Method is called when the mqtt client's corresponding websocket is closed. This method will send the subcribing clients to its private status an offline status message.

delMqttMosquittoClient ()

Method called after an mqtt clinet unsubsribes to atleast some topics, called by on_subscribe callback.

Returns Returns update mqqt clients active

Return type dict with update count

stop ()

Cleanly shutdown the connection to Mosquitto by disconnecting the mqtt client.

When mosquitto confirms disconection, on_disconnect callback will be called.

__dict__ = dict_proxy({'__module__': 'mosquittoChat.apps.mosquitto.pubsub', 'create_client': <function create_clien

__module__ = 'mosquittoChat.apps.mosquitto.pubsub'

__weakref__

list of weak references to the object (if defined)

Testing

NOTE: Testing is only done using the Docker Method. anyway, it should not matter whether you run your application using the Docker Method or the Standalone Method. Testing is independent of it.

Now, testing is the main deal of the project. You can test in many ways, namely, using `make` commands as mentioned in the below commands, which automates everything and you don't have to know anything else, like what test library or framework is being used, how the tests are happening, either directly or via `docker` containers, or may be different virtual environments using `tox`. Nothing is required to be known.

On the other hand if you want fine control over the tests, then you can run them directly, either by using `pytest` commands, or via `tox` commands to run them in different python environments or by using `docker-compose` commands to run differetn tests.

But running the make commands is lawasy the go to strategy and reccomended approach for this project.

NOTE: Tox can be used directly, where `docker` containers will not be used. Although we can try to run `tox` inside our test contianers that we are using for running the tests using the `make` commands, but then we would have to change the `Dockerfile` and install all the `python` dependencies like `python2.7`, `python3.x` and then run `tox` commands from inside the `docker` containers which then run the `pytest` commands which we run now to perform our tests inside the current test containers.

CAVEAT: The only caveat of using the make commands directly and not using `tox` is we are only testing the project in a single `python` environment, nameley `python 3.6`.

- To Test everything

  ```
  $ make test
  ```

  Any Other method without using make will involve writing a lot of commands. So use the make command preferrably

- To perform Unit Tests

  ```
  $ make test-unit
  ```

- To perform Component Tests

  ```
  $ make test-component
  ```

- To perform Contract Tests

  ```
  $ make test-contract
  ```

- To perform Integration Tests

  ```
  $ make test-integration
  ```

- To perform End To End (e2e) or System or UI Acceptance or Functional Tests

  ```
  $ make test-e2e

  # OR

  $ make test-system

  # OR

  $ make test-ui-acceptance
  ```

```
# OR

$ make test-functional
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index