# Table of Contents

1.1 Getting Started .......................................................... 4  
1.2 Frontend Console ......................................................... 4  
1.3 Monitoring Running Brokers .............................................. 4  
1.4 Advanced Configurations ................................................ 4  
1.5 Contribute ...................................................................... 4  
1.6 Project Roadmap ............................................................. 5  
1.7 References .................................................................... 6  
1.8 Licensing ....................................................................... 6  
1.9 Sandbox ........................................................................ 7
This project was originally developed Tegris Ltda for the FieldLink application and is distributed under the conditions described in the licensing chapter.

You may contact the project maintainer by dropping a message at @rafgoncalves or using the issue tracker.
1.1 Getting Started

1.1.1 Installation

1.1.2 Running the Broker

Command Line Options

1.1.3 Message Exchanging

1.1.4 Configuration Scripts

Supervisor

systemd

1.2 Frontend Console

1.2.1 Dashboards

Broker Dashboard

Session Dashboard

Topic Dashboard

1.3 Monitoring Running Brokers

1.3.1 Munin Integration

1.3.2 SNMP Interface

1.3.3 Amazon AWS Cloud Monitoring

1.4 Advanced Configurations

1.4.1 Session and Message Store Options

In Memory Storage
**Attention:** Check the open issues and pull requests to avoid reinventing the wheel. If you want to collaborate in solving an issue or developing a feature, please contact the person doing it through the issue tracker.

### 1.5.2 Bug Tracking

So you found a nasty bug?

Take a look at the [issue tracker](https://github.com/mqtt-broker/mqtt-broker/issues) and see if it was already reported. If you find a corresponding entry please upvote it and consider tackling the issue or helping those already involved. If you manage to solve the bug, open a pull request so we fix project for everyone.

If you found a previously unknown exemplar of software development fauna, please create a new entry and describe the bug as best as you can, possibly writing a unit test for it. Finally, state if you plan to solve it yourself or need help.

### 1.5.3 Documentation

You will find this documentation bundled with the source code, under the `docs` directory, in the project’s repository. It should be easy to compile it using the command:

```
you@your-pc:somewhere/mqtt-broker/docs$ make html
```

The results are built and stored at the `build` directory. The following steps are recommended:

1. Edit a page under `sources` directory;
2. Rebuild the html results (using the aforementioned command);
3. Check the results by opening `build/index.html`;
4. Repeat to your satisfaction;
5. Open a pull request so everyone can benefit from your work.

### 1.6 Project Roadmap

**Important:** These features are presented in no special order, neither is there a development schedule for them. If your project requires any of these features we encourage you to visit the corresponding BitBucket issue and express the situation.

You are also welcome to implement these and open a pull request.

#### 1.6.1 MQTT Compliance

1. MQTT Strict Mode [issue #1]
2. QoS Level 2, exactly once delivery [issue #2]
3. Last Will message [issue #3]
1.6.2 MQTT Protocol Extensions

1. Authentication support [issue #4]
2. Authorization support [issue #5]
3. Handling compressed publish messages (gzip) [issue #6]
4. Automatic compressing messages based on payload size [issue #7]

1.6.3 Integration Testing

1. Expand the test suite for QoS 0 [issue #8]
2. Expand the test suite for QoS 2 [issue #9]

1.6.4 Dependency Review

1. Consider Python AsyncIO instead of TornadoWeb [issue #10]

1.6.5 Message Store Support

1. File based message store [issue #11]
2. Redis backend [issue #12]

1.7 References

1.7.1 MQTT Protocol

1.7.2 Recommended MQTT Clients

1.7.3 Recommended MQTT Brokers

1.8 Licensing

1.8.1 An Open Source Effort and Contribution

This work is distributed as an contribution for the open source community, responsible for revolutionary tools as the Python Language, the Tornadoweb framework, the Sphinx-doc project and all the projects on which both this software and Tegris are build upon.

1.8.2 MIT License Terms

The MIT License (MIT) Copyright (c) 2014 Tegris LTDA

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:
1.9 Sandbox

```python
class broker.server.MQTTServer (authentication=None, persistence=None, clients=None, ssl_options=None)
```

This is the highest abstraction of the package and represents the whole MQTT Broker. It’s main roles are handling incoming connections, keeping tabs for the known client sessions and dispatching messages based on subscription matching.

**add_client** *(client)*

Register a client to the Broker.

**Parameters**

- **client** (*MQTTCient*) – A `broker.client.MQTTClient` instance.

**broadcast_message** *(msg)*

Broadcasts a message to all clients with matching subscriptions, respecting the subscription QoS.

**Parameters**

- **msg** (*Publish*) – A `broker.messages.Publish` instance.

**configure_last_will** *(client, connect_msg)*

Configures the last will message options for a given client on its connect message. Both the client and the connect message must point to the same client uid.

**Parameters**

- **MQTTCient** *(client)* – A client instance;
- **Connect** *(connect_msg)* – A Connect message that specifies the client.

**disconnect_all_clients** *

Disconnect all known clients.

**disconnect_client** *(client)*

Disconnects a MQTT client. Can be safely called without checking if the client is connected.

**Parameters**

- **MQTTCient** *(client)* – The MQTTCient to be disconnect

**dispatch_message** *(client, msg, cache=None)*

Dispatches a message to a client based on its subscriptions. It is safe to call this method without checking if the client has matching subscriptions.

**Parameters**

- **client** (*MQTTCient)* – The client which will possibly receive the message;
- **msg** (*Publish*) – The message to be delivered.
- **cache** (*dict*) – A dict that will be used for raw data caching. Defaults to a empty dictionary if None.

**enqueue_retained_message** *(client, subscription_mask)*

Enqueues all retained messages matching the `subscription_mask` to be sent to the `client`. 
Parameters

- **client** (*MQTTClient*) – A known MQTTClient.
- **subscription_mask** (*str*) – The subscription mask to match the messages against.

**get_known_client** (*connect_msg*)

Returns a known MQTTClient instance that has the same uid defined on the Connect message.

**Parameters**

- **Connect** (*connect_msg*) – A connect message that specifies the client.

**handle_incoming_publish** (*msg*)

Handles an incoming publish. This method is normally called by the clients a mechanism of notifying the server that there is a new message to be processed. The processing itself consists of retaining the message according with the *msg.retain* flag and broadcasting it to the subscribers.

**Parameters**

- **msg** (*Publish*) – The Publish message to be processed.

**handle_stream** (*stream, address*)

This coroutine is called by the Tornado loop whenever it receives a incoming connection. The server resolves the first message sent, checks if it’s a CONNECT frame and configures the client accordingly.

**Parameters**

- **stream** (*IOStream*) – A `tornado.iostream.IOStream` instance;
- **address** (*tuple*) – A tuple containing the ip and port of the connected client, ie ('127.0.0.1', 12345).

**remove_client** (*client*)

Removes a client from the know clients list. It’s safe to call this method without checking if the client is already known.

**Parameters**

- **client** (*MQTTClient*) – A `broker.client.MQTTClient` instance;

**Caution:** It won’t force client disconnection during the process, which can result in a lingering client in the Tornado loop.

**class** `broker.client.MQTTClient` (*server, connection, authorization=None, uid=None, clean_session=False, keep_alive=60, persistence=None*)

Objects of this class encapsulate and abstract all aspects of a given client. A MQTTClient object may refer to a live, connected, client or a known client which albeit disconnected had the clean_sessions flag set to false and, thus, is kept by the server as an end point for routed messages.

One may call `self.is_connected()` to check whether is there a connected client or not.

**Parameters**

- **server** (*MQTTServer*) – The server which the client is bound to;
- **connection** (*MQTTConnection*) – The connection to be used;
- **uid** (*str*) – A string used as the client’s id;
- **clean_session** (*bool*) – The clean session flag, as per MQTT Protocol;
- **keep_alive** (*int*) – The keep alive interval, in seconds.
- **persistence** (*ClientPersistenceBase*) – An object that provides persistence.
_on_connection_timeout (connection)
Callback called when the connection times out. Ensures clearing the self.connected event and processing the self.disconnect() method.

_on_stream_close (connection)
Callback called when the stream closes. Ensures clearing the self.connected event and processing the self.disconnect() method.

_process_incoming_packets (connection)
This coroutine fetches the message raw data from self.incoming_queue, parses it into the corresponding message object (an instance of one of the broker.messages.BaseMQTTMessage subclasses) and passes it to the self.incoming_transaction_manager to be processed.

It is started by calling self.start() and stops upon client disconnection.

_process_outgoing_packets (connection)
This coroutine fetches the message raw data from self.outgoing_queue, parses it into the corresponding message object (an instance of one of the broker.messages.BaseMQTTMessage subclasses) and passes it to the self.outgoing_transaction_manager to be processed.

It is started by calling self.start() and stops upon client disconnection.

configure_last_will (topic, payload, qos, retain=False)
Configures a message to be send as the client’s last will. This message will be send when the connected is disconnected by a connection timeout, protocol error or an unexpected disconnection.

Parameters

- topic (str) – The topic which the message will be delivered;
- payload (bytes) – Message payload;
- retain (bool) – Message’s retain flag.

closed
An toro.Event instance that is set whenever the client is connected and clear on disconnection. It’s safe to wait on this property before stream related operations.

disconnect ()
Closes the socket and disconnects the client. If self.clean_session is set, ensures that the incoming and outgoing queues are cleared and calls the server client removing routine.

Hint: It’s safe to call this function without checking whether the connection is open or not.

dispatch_to_server (pub_msg)
Dispatches a Publish message to the server for further processing, ie. delivering it to the appropriate subscribers.

Parameters pub_msg (Publish) – A broker.messages.Publish instance.

get_list_of_delivery_qos (msg)
Matches the msg.topic against all the current subscriptions and returns a list containing the QoS level for each matched subscription.

Parameters msg (Publish) – A MQTT valid message.

Return type tuple

Returns A list of QoS levels, ie [0, 0, 1, 2, 0, 2]

get_matching_qos (msg, subscriptions_mask)
Matches the msg.topic against a single subscription defined by the subscription mask and returns the QoS level on which the message should be delivered.
Parameters

- **msg** *(Publish)* – Message to be analysed;
- **subscriptions_mask** – A subscription mask that identifies one of the client’s subscriptions.

Returns QoS Level or None, in case it doesn’t match.

```python
handle_last_will()
```
Checks if a client has a pending last will message and dispatches it for server processing.

```python
is_connected()
```
A shorthand for :meth:`connected.is_set()`.

```python
publish(msg)
```
Puts a publish packet on the `self.outgoing_queue` to be sent to the client.

Parameters **msg** *(Publish)* – The message to be set or a iterable of its bytes.

```python
send_packet(packet)
```
Puts a packet on the `self.outgoing_queue` to be sent to the client.

```python
start()
```
Starts the client fetching, processing and dispatching routines. Should be called after object instantiation or a `self.update_connection()` call.

The following coroutines are started:

- `self._process_incoming_messages()`
- `self._process_outgoing_messages()`

```python
subscribe(subscription_mask, qos)
```
Subscribes the client to a topic or wildcarded mask at the informed QoS level. Calling this method also signalizes the server to enqueue the matching retained messages.

When called for a `(subscription_mask, qos)` pair for which the client has already a subscription it will silently ignore the command and return a suback.

Parameters

- **subscription_mask** *(string)* – A MQTT valid topic or wildcarded mask;
- **qos** *(int)* – A valid QoS level (0, 1 or 2).

Return type int

Returns The granted QoS level (0, 1 or 2) or 0x80 for failed subscriptions.

```python
unsubscribe(topics)
```
Unsubscribes the client from each topic in `topics`. Safely ignores topics which the client is not subscribed to.

Parameters **topics** *(iterable)* – An iterable of MQTT valid topic strings.

```python
update_configuration(clean_session=False, keep_alive=60)
```
Updates the internal attributes.

Parameters

- **clean_session** *(bool)* – A flag indicating whether this session should be brand new or attempt to reuse the last known session for a client with the same `self.uid` as this.
- **keep_alive** *(int)* – Connection’s keep alive setting, in seconds.
update_connection(connection)

Updates the client’s connection by disconnecting the previous one and configuring the new one’s keep alive time according to keep_alive.

Parameters connection (MQTTConnection) – The new connection to be used;

write(msg)

Writes a MQTT Message to the client. If the client isn’t connected, waits for the self.connected event to be set.

Parameters Message msg (MQTT) – The message to be send. It must be a instance of broker.messages.BaseMQTTMessage or it’s subclasses.
Symbols

_on_connection_timeout() (broker.client.MQTTClient method), 8
_on_stream_close() (broker.client.MQTTClient method), 9
_process_incoming_packets() (broker.client.MQTTClient method), 9
_process_outgoing_packets() (broker.client.MQTTClient method), 9

A
add_client() (broker.server.MQTTServer method), 7

B
broadcast_message() (broker.server.MQTTServer method), 7

C
configure_last_will() (broker.client.MQTTClient method), 9
get_known_client() (broker.server.MQTTServer method), 8
get_list_of_delivery_qos() (broker.client.MQTTClient method), 9
get_matching_qos() (broker.client.MQTTClient method), 9
connected (broker.client.MQTTClient attribute), 9

D
disconnect() (broker.client.MQTTClient method), 9
disconnect_all_clients() (broker.server.MQTTServer method), 7
disconnect_client() (broker.server.MQTTServer method), 7
dispatch_message() (broker.server.MQTTServer method), 7
dispatch_to_server() (broker.client.MQTTClient method), 9

E
enqueue_retained_message() (broker.server.MQTTServer method), 7

G
get_known_client() (broker.server.MQTTServer method), 8
get_list_of_delivery_qos() (broker.client.MQTTClient method), 9

H
handleIncoming_publish() (broker.server.MQTTServer method), 8
handle_last_will() (broker.client.MQTTClient method), 10
handle_stream() (broker.server.MQTTServer method), 8

I
is_connected() (broker.client.MQTTClient method), 10

M
MQTTClient (class in broker.client), 8
MQTTServer (class in broker.server), 7

P
publish() (broker.client.MQTTClient method), 10

R
remove_client() (broker.server.MQTTServer method), 8

S
send_packet() (broker.client.MQTTClient method), 10
start() (broker.client.MQTTClient method), 10
subscribe() (broker.client.MQTTClient method), 10

U
unsubscribe() (broker.client.MQTTClient method), 10
update_configuration() (broker.client.MQTTClient method), 10
update_connection() (broker.client.MQTTClient method), 10
W
write() (broker.client.MQTTClient method), 11