# aioamqp Documentation

***Release 0.5.0***

**Benoît Calvez**

December 22, 2015

Contents

Aioamqp is a library to connect to an amqp broker. It uses asyncio under the hood

CHAPTER 1

# Limitations

For the moment, aioamqp is tested against Rabbitmq.

Contents:

## 1.1 Introduction

This is the documentation for the aioamqp module.

### 1.1.1 Prerequisites

Aioamqp works only with python >= 3.3 using asyncio library. If your are using Python 3.3 you'll have to install asyncio from pypi, but asyncio is now included in python 3.4 standard library.

### 1.1.2 Installation

You can install the most recent aioamqp release from pypi using pip or easy_install:

```
pip install aioamqp
easy_install aioamqp
```

## 1.2 Changelog

### 1.2.1 Next version (not yet released)

- Add possibility to pass extra keyword arguments to protocol_factory when from_url is used to create a connection.
- Add SSL support.
- Support connection metadata customization, closes #40.
- Remove the use of rabbitmqctl in tests.
- Reduce the memory usage for channel recycling, closes #43.
- Add the usage of a previously created eventloop, closes #56.
- Removes the checks for coroutine callbacks, closes #55.

- Connection tuning are now configurable.
- Add a heartbeat method to know if the connection has fail, closes #3.
- Change the callback signature. It now takes the channel as first parameter, closes: #47.

### 1.2.2 Aioamqp 0.4.0

- Call the error callback on all circumtstances.

### 1.2.3 Aioamqp 0.3.0

- The consume callback takes now 3 parameters: body, envelope, properties, closes #33.
- Channel ids are now recycled, closes #36.

### 1.2.4 Aioamqp 0.2.1

- connect returns a transport and protocol instance.

### 1.2.5 Aioamqp 0.2.0

- Use a callback to consume messages.

## 1.3 API

### 1.3.1 Basics

There are two principal objects when using aioamqp:

- The protocol object, used to begin a connection to aioamqp,
- The channel object, used when creating a new channel to effectively use an AMQP channel.

### 1.3.2 Starting a connection

Starting a connection to AMQP really mean instanciate a new asyncio Protocol subclass.

aioamqp.**connect**(*host*, *port*, *login*, *password*, *virtualhost*, *ssl*, *login_method*, *insist*, *protocol_factory*, *verify_ssl*, *loop*, *kwargs*) → Transport, AmqpProtocol

Convenient method to connect to an AMQP broker

**Parameters**

- **host** (*str*) – the host to connect to
- **port** (*int*) – broker port
- **login** (*str*) – login
- **password** (*str*) – password
- **virtualhost** (*str*) – AMQP virtualhost to use for this connection

- **ssl** (*bool*) – Create an SSL connection instead of a plain unencrypted one
- **verify_ssl** (*bool*) – Verify server's SSL certificate (True by default)
- **login_method** (*str*) – AMQP auth method
- **insist** (*bool*) – Insist on connecting to a server
- **protocol_factory** (*AmqpProtocol*) – Factory to use, if you need to subclass AmqpProtocol
- **loop** (*EventLopp*) – Set the event loop to use
- **kwargs** (*dict*) – Arguments to be given to the protocol_factory instance

```
import asyncio
import aioamqp

@asyncio.coroutine
def connect():
    try:
        transport, protocol = yield from aioamqp.connect()  # use default parameters
    except aioamqp.AmqpClosedConnection:
        print("closed connections")
        return

    print("connected !")
    yield from asyncio.sleep(1)

    print("close connection")
    yield from protocol.close()
    transport.close()

asyncio.get_event_loop().run_until_complete(connect())
```

In this example, we just use the method "start_connection" to begin a communication with the server, which deals with credentials and connection tunning.

If you're not using the default event loop (e.g. because you're using aioamqp from a different thread), call aioamqp.connect(loop=your_loop).

The *AmqpProtocol* uses the *kwargs* arguments to configure the connection to the AMQP Broker:

**AmqpProtocol.__init__(self, *args, **kwargs):**

The protocol to communicate with AMQP

**Parameters**

- **channel_max** (*int*) – specifies highest channel number that the server permits. Usable channel numbers are in the range 1..channel-max. Zero indicates no specified limit.
- **frame_max** (*int*) – the largest frame size that the server proposes for the connection, including frame header and end-byte. The client can negotiate a lower value. Zero means that the server does not impose any specific limit but may reject very large frames if it cannot allocate resources for them.
- **heartbeat** (*int*) – the delay, in seconds, of the connection heartbeat that the server wants. Zero means the server does not want a heartbeat.
- **loop** (*Asyncio.EventLoop*) – specify the eventloop to use.
- **product** (*str*) – configure the client name product (like a UserAgent). product_version: str, configure the client product version.

### 1.3.3 Handling errors

The connect() method has an extra 'on_error' kwarg option. This on_error is a callback or a coroutine function which is called with an exception as the argument:

```
import asyncio
import aioamqp

@asyncio.coroutine
def error_callback(exception):
    print(exception)

@asyncio.coroutine
def connect():
    try:
        transport, protocol = yield from aioamqp.connect(
            host='nonexistant.com',
            on_error=error_callback,
        )
    except aioamqp.AmqpClosedConnection:
        print("closed connections")
        return

asyncio.get_event_loop().run_until_complete(connect())
```

### 1.3.4 Publishing messages

A channel is the main object when you want to send message to an exchange, or to consume message from a queue:

```
channel = yield from protocol.channel()
```

When you want to produce some content, you declare a queue then publish message into it:

```
queue = yield from channel.queue_declare("my_queue")
yield from queue.publish("aioamqp hello", '', "my_queue")
```

Note: we're pushing message to "my_queue" queue, through the default amqp exchange.

### 1.3.5 Consuming messages

When consuming message, you connect to the same queue you previously created:

```
import asyncio
import aioamqp

@asyncio.coroutine
def callback(body, envelope, properties):
    print(body)

channel = yield from protocol.channel()
yield from channel.basic_consume(callback, queue_name="my_queue")
```

The `basic_consume` method tells the server to send us the messages, and will call `callback` with amqp response arguments.

The `consumer_tag` is the id of your consumer, and the `delivery_tag` is the tag used if you want to acknowledge the message.

In the callback:

- the first `body` parameter is the message
- the `envelope` is an instance of envelope.Envelope class which encapsulate a group of amqp parameter such as:

```
consumer_tag
delivery_tag
exchange_name
routing_key
is_redeliver
```

- the `properties` are message properties, an instance of properties.Properties with the following members:

```
content_type
content_encoding
headers
delivery_mode
priority
correlation_id
reply_to
expiration
message_id
timestamp
type
user_id
app_id
cluster_id
```

### 1.3.6 Using exchanges

You can bind an exchange to a queue:

```
channel = yield from protocol.channel()
exchange = yield from channel.exchange_declare(exchange_name="my_exchange", type_name='fanout')
yield from channel.queue_declare("my_queue")
yield from channel.queue_bind("my_queue", "my_exchange")
```

CHAPTER 2

# Indices and tables

- genindex
- modindex
- search

## a

## A



## C