aioamqp Documentation

Release 0.13.0

Benoît Calvez

May 23, 2019

Contents

1 Limitations **3**
1.1 Introduction 3
1.2 API 3
1.3 Examples 10
1.4 Changelog 15

2 Indices and tables **19**

Python Module Index **21**

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

Aioamqp library is a pure-Python implementation of the AMQP 0.9.1 protocol using *asyncio*.

1.1.1 Prerequisites

Aioamqp works only with python >= 3.5 using asyncio library.

1.1.2 Installation

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

```
pip install aioamqp
```

1.2 API

1.2.1 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.

1.2.2 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

```python
import asyncio
import aioamqp

async def connect():
    try:
        transport, protocol = await aioamqp.connect()  # use default parameters
    except aioamqp.AmqpClosedConnection:
        print("closed connections")
        return

    print("connected !")
    await asyncio.sleep(1)

    print("close connection")
    await 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.
- **client_properties** (*dict*) – configure the client to connect to the AMQP server.

1.2.3 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:

```python
import asyncio
import socket
import aioamqp

async def error_callback(exception):
    print(exception)

async def connect():
    try:
        transport, protocol = await aioamqp.connect(
            host='nonexistant.com',
            on_error=error_callback,
            client_properties={
                'program_name': "test",
                'hostname' : socket.gethostname(),
            },

        )
    except aioamqp.AmqpClosedConnection:
        print("closed connections")
        return

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

1.2.4 Publishing messages

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

```python
channel = await protocol.channel()
```

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

```python
await channel.queue_declare("my_queue")
await channel.publish("aioamqp hello", '', "my_queue")
```

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

1.2.5 Consuming messages

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

```python
import asyncio
import aioamqp

async def callback(channel, body, envelope, properties):
    print(body)

channel = await protocol.channel()
await 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
  message_type
  user_id
  app_id
  cluster_id
  ```

Server Cancellation

RabbitMQ offers an AMQP extension to notify a consumer when a queue is deleted. See Consumer Cancel Notification for additional details. `aioamqp` enables the extension for all channels but takes no action when the consumer is cancelled. Your application can be notified of consumer cancellations by adding a callback to the channel:

```
async def consumer_cancelled(channel, consumer_tag):
    # implement required cleanup here
    pass


async def consumer(channel, body, envelope, properties):
    await channel.basic_client_ack(envelope.delivery_tag)


channel = await protocol.channel()
channel.add_cancellation_callback(consumer_cancelled)
await channel.basic_consume(consumer, queue_name="my_queue")
```

The callback can be a simple callable or an asynchronous co-routine. It can be used to restart consumption on the channel, close the channel, or anything else that is appropriate for your application.

1.2.6 Queues

Queues are managed from the *Channel* object.

`Channel.`**`queue_declare`**(*queue_name*, *passive*, *durable*, *exclusive*, *auto_delete*, *no_wait*, *arguments*, *timeout*) → dict

Coroutine, creates or checks a queue on the broker

Parameters

- **queue_name** (*str*) – the queue to receive message from
- **passive** (*bool*) – if set, the server will reply with *Declare-Ok* if the queue already exists with the same name, and raise an error if not. Checks for the same parameter as well.
- **durable** (*bool*) – if set when creating a new queue, the queue will be marked as durable. Durable queues remain active when a server restarts.
- **exclusive** (*bool*) – request exclusive consumer access, meaning only this consumer can access the queue
- **no_wait** (*bool*) – if set, the server will not respond to the method
- **arguments** (*dict*) – AMQP arguments to be passed when creating the queue.
- **timeout** (*int*) – wait for the server to respond after *timeout*

Here is an example to create a randomly named queue with special arguments *x-max-priority*:

```
result = await channel.queue_declare(
    queue_name='', durable=True, arguments={'x-max-priority': 4}
)
```

`Channel.`**`queue_delete`**(*queue_name*, *if_unused*, *if_empty*, *no_wait*, *timeout*)

Coroutine, delete a queue on the broker

Parameters

- **queue_name** (*str*) – the queue to receive message from
- **if_unused** (*bool*) – the queue is deleted if it has no consumers. Raise if not.
- **if_empty** (*bool*) – the queue is deleted if it has no messages. Raise if not.
- **no_wait** (*bool*) – if set, the server will not respond to the method

- **arguments** (*dict*) – AMQP arguments to be passed when creating the queue.
- **timeout** (*int*) – wait for the server to respond after *timeout*

Channel.**queue_bind**(*queue_name*, *exchange_name*, *routing_key*, *no_wait*, *arguments*, *timeout*)
Coroutine, bind a *queue* to an *exchange*

Parameters

- **queue_name** (*str*) – the queue to receive message from.
- **exchange_name** (*str*) – the exchange to bind the queue to.
- **routing_key** (*str*) – the routing_key to route message.
- **no_wait** (*bool*) – if set, the server will not respond to the method
- **arguments** (*dict*) – AMQP arguments to be passed when creating the queue.
- **timeout** (*int*) – wait for the server to respond after *timeout*

This simple example creates a *queue*, an *exchange* and bind them together.

```
channel = await protocol.channel()
await channel.queue_declare(queue_name='queue')
await channel.exchange_declare(exchange_name='exchange')

await channel.queue_bind('queue', 'exchange', routing_key='')
```

Channel.**queue_unbind**(*queue_name*, *exchange_name*, *routing_key*, *arguments*, *timeout*)
Coroutine, unbind a queue and an exchange.

Parameters

- **queue_name** (*str*) – the queue to receive message from.
- **exchange_name** (*str*) – the exchange to bind the queue to.
- **no_wait** (*bool*) – if set, the server will not respond to the method
- **arguments** (*dict*) – AMQP arguments to be passed when creating the queue.
- **timeout** (*int*) – wait for the server to respond after *timeout*

PARAM STR ROUTING_KEY THE ROUTING_KEY TO ROUTE MESSAGE.

Channel.**queue_purge**(*queue_name*, *no_wait*, *timeout*)
Coroutine, purge a queue

Parameters **queue_name** (*str*) – the queue to receive message from.

1.2.7 Exchanges

Exchanges are used to correctly route message to queue: a *publisher* publishes a message into an exchanges, which routes the message to the corresponding queue.

Channel.**exchange_declare**(*exchange_name*, *type_name*, *passive*, *durable*, *auto_delete*, *no_wait*, *arguments*, *timeout*) → dict
Coroutine, creates or checks an exchange on the broker

Parameters

- **exchange_name** (*str*) – the exchange to receive message from
- **type_name** (*str*) – the exchange type (fanout, direct, topics . . .)

- **passive** (*bool*) – if set, the server will reply with *Declare-Ok* if the exchange already exists with the same name, and raise an error if not. Checks for the same parameter as well.
- **durable** (*bool*) – if set when creating a new exchange, the exchange will be marked as durable. Durable exchanges remain active when a server restarts.
- **auto_delete** (*bool*) – if set, the exchange is deleted when all queues have finished using it.
- **no_wait** (*bool*) – if set, the server will not respond to the method
- **arguments** (*dict*) – AMQP arguments to be passed when creating the exchange.
- **timeout** (*int*) – wait for the server to respond after *timeout*

Note: the *internal* flag is deprecated and not used in this library.

```
channel = await protocol.channel()
await channel.exchange_declare(exchange_name='exchange', auto_delete=True)
```

Channel.**exchange_delete**(*exchange_name*, *if_unused*, *no_wait*, *timeout*)
Coroutine, delete a exchange on the broker

Parameters

- **exchange_name** (*str*) – the exchange to receive message from
- **if_unused** (*bool*) – the exchange is deleted if it has no consumers. Raise if not.
- **no_wait** (*bool*) – if set, the server will not respond to the method
- **arguments** (*dict*) – AMQP arguments to be passed when creating the exchange.
- **timeout** (*int*) – wait for the server to respond after *timeout*

Channel.**exchange_bind**(*exchange_destination*, *exchange_source*, *routing_key*, *no_wait*, *arguments*, *timeout*)
Coroutine, binds two exchanges together

Parameters

- **exchange_destination** (*str*) – specifies the name of the destination exchange to bind
- **exchange_source** (*str*) – specified the name of the source exchange to bind.
- **exchange_destination** – specifies the name of the destination exchange to bind
- **no_wait** (*bool*) – if set, the server will not respond to the method
- **arguments** (*dict*) – AMQP arguments to be passed when creating the exchange.
- **timeout** (*int*) – wait for the server to respond after *timeout*

Channel.**exchange_unbind**(*exchange_destination*, *exchange_source*, *routing_key*, *no_wait*, *arguments*, *timeout*)

Coroutine, unbind an exchange from an exchange.

Parameters

- **exchange_destination** (*str*) – specifies the name of the destination exchange to bind
- **exchange_source** (*str*) – specified the name of the source exchange to bind.
- **exchange_destination** – specifies the name of the destination exchange to bind

- **no_wait** (*bool*) – if set, the server will not respond to the method
- **arguments** (*dict*) – AMQP arguments to be passed when creating the exchange.
- **timeout** (*int*) – wait for the server to respond after *timeout*

1.3 Examples

Those examples are ported from the RabbitMQ tutorial. They are specific to *aioamqp* and uses *coroutines* exclusievely. Please read both documentation together, as the official documentation explain how to use the AMQP protocol correctly.

Do not hesitate to use RabbitMQ Shiny management interfaces, it really helps to understand which message is stored in which queues, and which consumer unqueues what queue.

Using docker, you can run RabbitMQ using the following command line. Using this command line you will be able to run the examples and access the RabbitMQ management interface using the login *guest* and the password *guest*.

Contents:

1.3.1 "Hello World!" : The simplest thing that does something

Sending

Our first script to send a single message to the queue.

Creating a new connection:

```
import asyncio
import aioamqp

async def connect():
    transport, protocol = await aioamqp.connect()
    channel = await protocol.channel()

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

This first scripts shows how to create a new connection to the *AMQP* broker.

Now we have to declare a new queue to receive our messages:

```
await channel.queue_declare(queue_name='hello')
```

We're now ready to publish message on to this queue:

```
await channel.basic_publish(
    payload='Hello World!',
    exchange_name='',
    routing_key='hello'
)
```

We can now close the connection to rabbit:

```
# close using the `AMQP` protocol
await protocol.close()
# ensure the socket is closed.
transport.close()
```

You can see the full example in the file *example/send.py*.

Receiving

We now want to unqueue the message in the consumer side.

We have to ensure the queue is created. Queue declaration is indempotant.

```
await channel.queue_declare(queue_name='hello')
```

To consume a message, the library calls a callback (which **MUST** be a coroutine):

```
async def callback(channel, body, envelope, properties):
    print(body)

await channel.basic_consume(callback, queue_name='hello', no_ack=True)
```

1.3.2 Work Queues : Distributing tasks among workers

The main purpose of this part of the tutorial is to *ack* a message in RabbitMQ only when it's really processed by a worker.

new_task

This publisher creates a queue with the *durable* flag and publish a message with the property *persistent*.

```
await channel.queue('task_queue', durable=True)

await channel.basic_publish(
    payload=message,
    exchange_name='',
    routing_key='task_queue',
    properties={
        'delivery_mode': 2,
    },
)
```

worker

The purpose of this worker is to simulate a resource consuming execution which delays the processing of the other messages.

The worker declares the queue with the exact same argument of the *new_task* producer.

```
await channel.queue('task_queue', durable=True)
```

Then, the worker configure the *QOS*: it specifies how the worker unqueues message.

```
await channel.basic_qos(prefetch_count=1, prefetch_size=0, connection_
↪global=False)
```

Finaly we have to create a callback that will *ack* the message to mark it as *processed*. Note: the code in the callback calls *asyncio.sleep* to simulate an asyncio compatible task that takes time. You probably want to block the eventloop to simulate a CPU intensive task using *time.sleep*.

```
async def callback(channel, body, envelope, properties):
    print(" [x] Received %r" % body)
    await asyncio.sleep(body.count(b'.'))
    print(" [x] Done")
    await channel.basic_client_ack(delivery_tag=envelope.delivery_tag)
```

1.3.3 Publish Subscribe : Sending messages to many consumers at once

This part of the tutorial introduce *exchange*.

A *emit_log.py* scripts publish messages into a *fanout* exchange. Then the *receive_log.py* script creates a temporary queue (which is deleted on the disconnection).

If the script *receive_log.py* is ran multiple times, all the instance will receive the message emitted by *emit_log*.

Publisher

The publisher create a new *fanout* exchange:

```
await channel.exchange_declare(exchange_name='logs', type_name='fanout')
```

And publish message into that exchange:

```
await channel.basic_publish(message, exchange_name='logs', routing_key='')
```

Consumer

The consumer create a temporary queue and binds it to the exchange.

```
await channel.exchange(exchange_name='logs', type_name='fanout')
# let RabbitMQ generate a random queue name
result = await channel.queue(queue_name='', exclusive=True)

queue_name = result['queue']
await channel.queue_bind(exchange_name='logs', queue_name=queue_name,␣
↪routing_key='')
```

1.3.4 Routing : Receiving messages selectively

Routing is an interesting concept in RabbitMQ/AMQP: in this tutorial, messages are published to a *direct* exchange with a specific routing_key (the log *severity* The *consumer* create a queue, binds the queue to the exchange and specifies the severity he wants to receive.

Publisher

The publisher creater the *direct* exchange:

```
await channel.exchange(exchange_name='direct_logs', type_name='direct')
```

Message are published into that exchange and routed using the severity for instance:

```
await channel.publish(message, exchange_name='direct_logs', routing_key='info
↪')
```

Consumer

The consumer may subscribe to multiple severities. To accomplish this purpose, it create a queue bind this queue multiple time using the *(exchange_name, routing_key)* configuration:

```
result = await channel.queue(queue_name='', durable=False, auto_delete=True)

queue_name = result['queue']

severities = sys.argv[1:]
if not severities:
    print("Usage: %s [info] [warning] [error]" % (sys.argv[0],))
    sys.exit(1)

for severity in severities:
    await channel.queue_bind(
        exchange_name='direct_logs',
        queue_name=queue_name,
        routing_key=severity,
    )
```

1.3.5 Topics : Receiving messages based on a pattern

Topics are another exchange type. It allows message routing depending on a pattern, to route a message for multiple criteria. We're going to use a topic exchange in our logging system. We'll start off with a working assumption that the routing keys of logs will have two words: "<facility>.<severity>".

Publisher

The publisher prepares the exchange and publish messages using a routing_key which will be matched by later filters

```
await channel.exchange('topic_logs', 'topic')

await await channel.publish(message, exchange_name=exchange_name, routing_
↪key='anonymous.info')
await await channel.publish(message, exchange_name=exchange_name, routing_
↪key='kern.critical')
```

Consumer

The consumer selects the combination of 'facility'/'severity' he wants to subscribe to:

```
for binding_key in ("*.critical", "nginx.*"):
    await channel.queue_bind(
        exchange_name='topic_logs',
        queue_name=queue_name,
        routing_key=binding_key
    )
```

1.3.6 RPC: Remote procedure call implementation

This tutorial will try to implement the RPC as in the RabbitMQ's tutorial.

The API will probably look like:

```
fibonacci_rpc = FibonacciRpcClient()
result = await fibonacci_rpc.call(4)
print("fib(4) is %r" % result)
```

Client

In this case it's no more a producer but a Client: we will send a message in a queue and wait for a response in another. For that purpose, we publish a message to the *rpc_queue* and add a *reply_to* properties to let the server know where to respond.

```
result = await channel.queue_declare(exclusive=True)
callback_queue = result['queue']

channel.basic_publish(
    exchange='',
    routing_key='rpc_queue',
    properties={
        'reply_to': callback_queue,
    },
    body=request,
)
```

Note: the client use a *waiter* (an asyncio.Event) which will be set when receiving a response from the previously sent message.

Server

When unqueing a message, the server will publish a response directly in the callback. The *correlation_id* is used to let the client know it's a response from this request.

```
async def on_request(channel, body, envelope, properties):
    n = int(body)

    print(" [.] fib(%s)" % n)
    response = fib(n)

    await channel.basic_publish(
        payload=str(response),
        exchange_name='',
        routing_key=properties.reply_to,
```

(continues on next page)

(continued from previous page)

```
        properties={
            'correlation_id': properties.correlation_id,
        },
    )

    await channel.basic_client_ack(delivery_tag=envelope.delivery_tag)
```

1.4 Changelog

1.4.1 Next release

- Rename `type` to `message_type` in constant.Properties object to be full compatible with pamqp.

1.4.2 Aioamqp 0.13.0

- SSL Connections must be configured with an SSLContext object in `connect` and `from_url` (closes #142).
- Uses pamqp to encode or decode protocol frames.
- Drops support of python 3.3 and python 3.4.
- Uses async and await keywords.
- Fix pamqp *_frame_parts* call, now uses exposed *frame_parts*

1.4.3 Aioamqp 0.12.0

- Fix an issue to use correct int encoder depending on int size (closes #180).
- Call user-specified callback when a consumer is cancelled.

1.4.4 Aioamqp 0.11.0

- Fix publish str payloads. Support will be removed in next major release.
- Support for `basic_return` (closes #158).
- Support for missings encoding and decoding types (closes #156).

1.4.5 Aioamqp 0.10.0

- Remove `timeout` argument from all channel methods.
- Clean up uses of `no_wait` argument from most channel methods.
- Call `drain()` after sending every frame (or group of frames).
- Make sure AmqpProtocol behaves identically on 3.4 and 3.5+ wrt EOF reception.

1.4.6 Aioamqp 0.9.0

- Fix server cancel handling (closes #95).
- Send "close ok" method on server-initiated close.
- Validate internal state before trying to send messages.
- Clarify which BSD license we actually use (3-clause).

1.4.7 Aioamqp 0.8.2

- Really turn off heartbeat timers (closes #112).

1.4.8 Aioamqp 0.8.1

- Turn off heartbeat timers when the connection is closed (closes #111).
- Fix tests with python 3.5.2 (closes #107).
- Properly handle unlimited sized payloads (closes #103).
- API fixes in the documentation (closes #102, #110).
- Add frame properties to returned value from `basic_get()` (closes #100).

1.4.9 Aioamqp 0.8.0

- Complete overhaul of heartbeat (closes #96).
- Prevent closing channels multiple times (inspired by PR #88).

1.4.10 Aioamqp 0.7.0

- Add `basic_client_nack` and `recover` method (PR #72).
- Sends `server-close-ok` in response to a `server-close`.
- Disable Nagle algorithm in `connect` (closes #70).
- Handle `CONNECTION_CLOSE` during initial protocol handshake (closes #80).
- Supports for python 3.5.
- Few code refactors.
- Dispatch does not catch `KeyError` anymore.

1.4.11 Aioamqp 0.6.0

- The `client_properties` is now fully configurable.
- Add more documentation.
- Simplify the channel API: `queue_name` arg is no more required to declare a queue. `basic_qos` arguments are now optional.

1.4.12 Aioamqp 0.5.1

- Fixes packaging issues when uploading to pypi.

1.4.13 Aioamqp 0.5.0

- 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.

1.4.14 Aioamqp 0.4.0

- Call the error callback on all circumtstances.

1.4.15 Aioamqp 0.3.0

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

1.4.16 Aioamqp 0.2.1

- connect returns a transport and protocol instance.

1.4.17 Aioamqp 0.2.0

- Use a callback to consume messages.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

aioamqp, 3

Index

A

`aioamqp` (*module*), 3

C

`connect()` (*in module aioamqp*), 4

E

`exchange_bind()` (*aioamqp.Channel method*), 9
`exchange_declare()` (*aioamqp.Channel method*), 8
`exchange_delete()` (*aioamqp.Channel method*), 9
`exchange_unbind()` (*aioamqp.Channel method*), 9

Q

`queue_bind()` (*aioamqp.Channel method*), 8
`queue_declare()` (*aioamqp.Channel method*), 7
`queue_delete()` (*aioamqp.Channel method*), 7
`queue_purge()` (*aioamqp.Channel method*), 8
`queue_unbind()` (*aioamqp.Channel method*), 8