# RbFly Library

***Release 0.10.0***

**Artur Wroblewski**

**Oct 17, 2024**

## RbFly Library

Part I

# RbFly Library

# 1 Overview

RbFly is a library for RabbitMQ Streams using Python asyncio.

## 1.1 Features

The library is designed and implemented with the following qualities in mind

1. Simple, flexible, and asynchronous Pythonic API with type annotations.
2. Use of AMQP 1.0 message format to enable interoperability between RabbitMQ Streams clients.
3. Performance. For example, RbFly can be over 4 times faster than other, similar solutions when publishing messages to a streaming broker (Section 1.4 references appropriate report).
4. Auto reconnection to RabbitMQ broker with lazily created connection objects.

RbFly supports many RabbitMQ Streams broker features

1. Publishing single messages, or in batches, with confirmation.
2. Subscribing to a stream at a specific point in time, from a specific offset, or using offset reference.
3. Stream message filtering.
4. Writing stream offset reference.
5. Message deduplication.
6. Integration with AMQP 1.0 ecosystem at message format level.

## 1.2 Requirements

RbFly requires

- Python 3.11, or later
- RabbitMQ 3.13.0, or later
- Cython 3.0.0 (for development)

## 1.3 Installation

Install or upgrade RbFly from PyPi with pip command:

```
$ pip install -U rbfly
```

## 1.4 Documentation

The documentation of RbFly library is hosted at project's website at the address

https://wrobell.dcmod.org/rbfly/

The documentation consists of multiple parts. Each part can be downloaded in PDF format

| Document | URL |
|---|---|
| RbFly Library | https://wrobell.dcmod.org/rbfly/rbfly-0.10.0.pdf |
| RbFly Library Performance Report | https://wrobell.dcmod.org/rbfly/rbfly-performance-0.10.0.pdf |

## 1.5 Links

1. RbFly project website: https://wrobell.dcmod.org/rbfly/index.html
2. Report bugs for RbFly project: https://gitlab.com/wrobell/rbfly/-/issues
3. View RbFly library source code: https://gitlab.com/wrobell/rbfly

4. RabbitMQ Streams: https://www.rabbitmq.com/streams.html
5. RabbitMQ Streams Java library: https://rabbitmq.github.io/rabbitmq-stream-java-client/stable/htmlsingle/
6. RabbitMQ discussions: https://groups.google.com/g/rabbitmq-users/

### 1.6 Acknowledgements

The design and implementation of RbFly library is inspired by other projects

- asyncpg
- rstream
- aioredis
- aiokafka

## 2 License



### 2.1 Proprietary License



## 3 Quick Start

The example below demonstrates various RbFly library features, which can be utilized in an application using RabbitMQ Streams:

- creating RabbitMQ Streams client using connection URI
- creating a stream (*demo* coroutine)
- creating a stream publisher and publishing messages to a stream (*send_data* coroutine)
- subscribing to a stream and receiving messages from a stream (*receive_data* coroutine)
- the script continues to run if RabbitMQ Streams broker stops and starts again

```
import asyncio
from datetime import datetime

import rbfly.streams as rbs

STREAM = 'rbfly-demo-stream'  # stream to send message to

async def send_data(client: rbs.StreamsClient) -> None:
    # create publisher and send messages to the stream in a loop
    async with client.publisher(STREAM) as publisher:
        while True:
            await publisher.send('hello')

            print('{} message sent'.format(datetime.now()))
            await asyncio.sleep(5)
```

(continues on next page)

(continued from previous page)

```
async def receive_data(client: rbs.StreamsClient) -> None:
    # subscribe to the stream and receive the messages
    async for msg in client.subscribe(STREAM):
        print('{} got: {!r}'.format(datetime.now(), msg))
        print()

@rbs.connection  # close connection to RabbitMQ Streams broker at exit
async def demo(client: rbs.StreamsClient) -> None:
    # create stream; operation does nothing if a stream exists; stream can
    # be created by an external tool;
    # this is first operation to RabbitMQ Streams broker in this demo,
    # so a connection is created by RbFly
    await client.create_stream(STREAM)

    await asyncio.gather(send_data(client), receive_data(client))

# create RabbitMQ Streams client
client = rbs.streams_client('rabbitmq-stream://guest:guest@localhost')

asyncio.run(demo(client))
```

The source code of the demo can be downloaded from RbFly code repository.

The following sections of the documentation discuss the features of RbFly library in more detail.

# 4 Publish Messages

To send messages to a RabbitMQ stream create a RabbitMQ Streams publisher with *publisher()* method of *StreamsClient()* class. There are multiple types of publishers implemented by the following classes

***rbfly.streams.Publisher***
: Send a message and wait for RabbitMQ Streams broker to confirm that the message is received. Multiple asynchronous coroutines can send messages with the same publisher concurrently. This is the slowest way of sending messages. It is the default RbFly publisher.

***rbfly.streams.PublisherBatchLimit***
: Batch multiple messages, then send them to a stream with the flush method. Batch method blocks when a limit of messages is reached, so an application does not run of out memory. Batching and flushing can be performed concurrently from different asynchronous coroutines. This is faster method of sending messages, than the previous one.

***rbfly.streams.PublisherBatchFast***
: Batch multiple messages, then send them to a stream with the flush method. The number of messages in a batch is limited by a maximum length of Python list. An application needs to flush messages on a regular basis to sustain its performance, and to avoid running out of memory. Use it only when messages can be batched and flushed in a sequence, i.e. from the same asynchronous coroutine. This is the fastest method of sending messages to a stream, but provides no coordination between batch and flush methods.

**Note**

RabbitMQ Streams publisher uses publisher reference name for message deduplication. By default, a publisher name is set using <hostname>/<pid> format.

If the naming scheme is not sufficient, then publisher reference name should be overriden when creating a publisher.

## 4.1 Send Single Message

To send messages one by one, create a RabbitMQ Streams publisher and send messages with *send()* asynchronous coroutine:

```
async with client.publisher('stream-name') as publisher:
    message = 'hello'
    await publisher.send(message)
```

The coroutine sends a message and waits for RabbitMQ Streams broker for the published message confirmation.

## 4.2 Send Batch of Messages

If an application has to send a batch of messages from multiple asynchronous coroutines, then use publisher implemented by *PublisherBatchLimit* class.

Enqueue a message with *batch()* asynchronous coroutine method. The method blocks if an applications reaches limit of number of messages. To unblock, use *flush()* asynchronous coroutine method:

```
async def run_app(client: StreamsClient) -> None:
    # note: publisher is flushed on exit of the context manager as well
    async with client.publisher('stream-name', cls=PublisherBatchLimit) as publisher:
        await asyncio.gather(batch(publisher), flush(publisher))

async def batch(publisher: PublisherBatchLimit) -> None:
    await publisher.batch('hello 1')
    await publisher.batch('hello 2')

async def flush(publisher: PublisherBatchLimit) -> None:
    while True:
        await asyncio.sleep(0.2)  # flush messages 5 times per second
        await publisher.flush()
```

Use *flush()* asynchronous coroutine to send messages to RabbitMQ Streams broker and wait for confirmation of receiving the messages.

## 4.3 Fast Batch Publishing

To send messages in batch mode create publisher using *PublisherBatchFast* class. Enqueue each message with *batch()* method:

```
async with client.publisher('stream-name', cls=PublisherBatchFast) as publisher:
    publisher.batch('hello 1')
    publisher.batch('hello 2')
    await publisher.flush()
```

Use *flush()* asynchronous coroutine to send the messages to RabbitMQ Streams broker and wait for confirmation of receiving the messages.

## 4.4 Publishing Exceptions

RbFly might raise the following exceptions when publishing messages

**TypeError**
: A message or part of a message is of a type not recognized by RbFly's AMQP format encoder.

**ValueError**
: A message has invalid value. For example, a message does not fit within the frame of RabbitMQ Streams protocol, or a string exceeds maximum allowed length (`2 ** 32 - 1`).

# 5 Read Messages

## 5.1 Subscribe to a Stream

To read messages from RabbitMQ Streams broker, subscribe to a RabbitMQ stream with *subscribe()* method of *StreamsClient* class. The method is an asynchronous iterator, which yields stream messages:

```
async for msg in client.subscribe('stream-name'):
    print(msg)
```

The method accepts the offset parameter, which is set to *rbfly.streams.Offset.NEXT* by default. Stream offsets are described in Section 5.3 in more detail.

## 5.2 Message Context

A RabbitMQ Streams message has a set of properties like offset or timestamp. Also, AMQP 1.0 message, beside message body, can have additional data attached to it, for example message header or application properties.

The additional information is available via *MessageCtx* object, which can be retrieved with *get_message_ctx()* function, for example:

```
async for msg in client.subscribe('stream-name'):
    print(get_message_ctx().stream_offset)
```

## 5.3 Offset Specification

Use RabbitMQ Streams offset specification to declare which messages an application should receive from a stream.

The *subscribe()* method accepts optional *offset* parameter, which can be one of the following:

***rbfly.streams.Offset.NEXT***
: Receive new messages from a stream only. Default offset specification.

***rbfly.streams.Offset.FIRST***
: Receive all messages, starting with the very first message in a stream. Equivalent to *Offset.offset(0)*.

***rbfly.streams.Offset.LAST***
: Receive messages from a streams starting with first message stored in the current stream chunk (see also below).

***rbfly.streams.Offset.offset()***
: Receive messages from a stream starting with specific offset value.

***rbfly.streams.Offset.reference()***
: Use the reference to get the offset stored in a stream. Receive messages starting from the next offset (this is *offset + 1*).

***rbfly.streams.Offset.timestamp()***
: Receive messages from a stream starting with the specified timestamp of a message.

The following diagram visualizes offset location in a stream when each chunk has 100 messages:

```
                        +- Offset.reference('ref-a') + 1
                        |
chunk 1: [0] [1, ref-a] [2] ... [99]
         |
         +- Offset.FIRST

chunk 2: [100, 1633006475.571] [101, 1633006475.999] ... [199, 1633006477.999]
                                |                           |
                                +- Offset.offset(101)       +- Offset.
↪timestamp(1633006476.0)
```

(continues on next page)

(continued from previous page)

```
...    : ...
                                  +- end of stream
                                  |
chunk 10: [900] [901] ... [999]   +
            |                     |
            +- Offset.LAST        +- Offset.NEXT
```

**Note**

Timestamp is Erlang runtime system time. It is a view of POSIX time.

### 5.4 Offset Reference

RabbitMQ Streams supports storing an offset value in a stream using a string reference, and receiving an offset value using the reference.

Use *offset reference specification* to read messages from a stream starting after stored offset value:

```
messages = client.subscribe('stream-name', offset=Offset.reference('stream-ref'))
try:
    async for msg in messages:
        print msg
finally:
    await client.write_offset('stream-name', 'stream-ref')
```

In the example above, RbFly library performs the following actions

- read offset value, stored for offset reference string *stream-ref*, from RabbitMQ Streams broker
- start reading messages of stream *stream-name* with the offset value increased by one (start reading *after* stored value)
- keep offset value of last received message in memory

The example saves offset value with *write_offset()* method. By default, the method saves offset value of last stream message. Custom offset value, can also be specified, see *the method documentation* for details.

**Note**

How and when offset value shall be saved is application dependant.

## 6 Manage Connection

Use *streams_client()* function to declare connection to RabbitMQ Streams broker. The function creates RabbitMQ Streams client object, which is used to *create streams*, *publish messages* to a stream, and *subscribe* to a stream.

Whenever an action of an application requires an interaction with RabbitMQ Streams broker, a connection is created, or existing connection is reused.

When a broker is restarted (i.e. after upgrade, or after crash), then RbFly reconnects the client, and retries an interrupted action.

Properly closing a connection to RabbitMQ Streams broker requires use of *connection()* decorator. Use it to decorate a Python asynchronous coroutine function. The decorator closes a connection to a broker when the decorated coroutine exits.

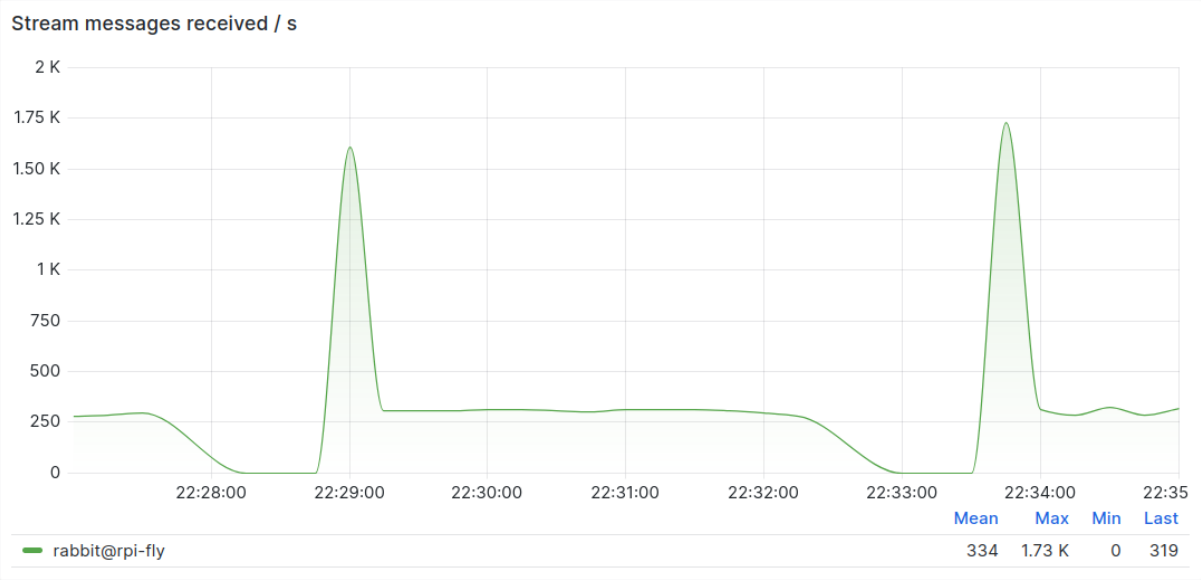


Figure 6.1: RbFly reconnecting an application to RabbitMQ Streams broker in action
Screenshot of Grafana plot of stream messages received by RabbitMQ Streams broker from an application. There are two restarts of RabbitMQ service to update the broker. Each time, RbFly receives disconnection request from the broker, caches messages of an application, and flushes messages to the broker when a new connection is established. No data is lost in the process.

The example below is a skeleton of a Python program creating RabbitMQ Streams broker client, implementing asynchronous coroutine to interact with the broker, and to close connection on the coroutine exit:

```
import asyncio
import rbfly.streams as rbs

@rbs.connection
async def streams_app(client: rbs.StreamsClient) -> None:
    ...
    # interaction with RabbitMQ Streams broker
    ...

client = rbs.streams_client('rabbitmq-stream://guest:guest@localhost')
asyncio.run(streams_app(client))
```

# 7 Deduplicate Messages

RabbitMQ Streams allows applications to publish messages to a stream with message deduplication.

RbFly generates message publishing ids by default. It also supports message deduplication, but it is responsibility of an application to

- assign reference name to a publisher of messages
- assign a publishing id to a message in an unique fashion
- ensure publishing ids of messages are strictly increasing
- keep track of publishing id of a message between application restarts

**Note**

A RbFly publisher object remembers publishing id of a last message. Therefore, it is possible to switch be-

tweeen providing a message publishing id by an application, and generating message publishing id by RbFly library.

However, it is not recommended for an application to change the approach to message publishing ids during application lifetime.

## 7.1 Publish Messages with Id

To enable deduplication of messages create a publisher with an unique reference name for given application. Create context for each message with *stream_message_ctx()* function, and publish message context with a publisher.

The following example publishes *10* messages every second. Every time, the messages are assigned the same message publishing id. The number of messages in the stream is always *10*, even when the script is restarted.

```
import asyncio
from datetime import datetime

import rbfly.streams as rbs

STREAM = 'rbfly-demo-dedup-stream'

# an application uses an unique publisher name; it allows message
# deduplication between application restarts
PUBLISHER = 'demo-publisher'

async def send_data(client: rbs.StreamsClient) -> None:
    async with client.publisher(STREAM, name=PUBLISHER) as publisher:
        while True:
            for i in range(10):
                # use `stream_message_ctx` function to create message
                # context and assign publishing id
                ctx = rbs.stream_message_ctx('hello', publish_id=i)
                await publisher.send(ctx)

            print('{} messages sent'.format(datetime.now()))
            await asyncio.sleep(1)

@rbs.connection
async def demo(client: rbs.StreamsClient) -> None:
    await client.create_stream(STREAM)
    await send_data(client)

client = rbs.streams_client('rabbitmq-stream://guest:guest@localhost')
asyncio.run(demo(client))
```

**Note**

Observe number of messages in a RabbitMQ stream with *rabbitmqctl list_queues* command, or in the management web console.

When an application is restarted, and a publisher is created with its reference name, then RbFly library queries RabbitMQ Streams broker to get the last message publishing id. A RbFly's publisher object provides its last message publishing id via *message_id* attribute.

## 7.2 Order of Messages

It is an application's responsibility to publish messages with their publishing id strictly increasing. Providing out-of-order message publishing ids has an undefined result.

When publishing a single message at a time (see Section 4.1), then publishing id, of each new message, has to have greater value comparing to a previous one. When existing messages are sent to a broker, then the order does not matter - messages are ignored due to RabbitMQ Stream's deduplication feature.

For new messages, the same rule as above applies to a batch publisher. However, when republishing messages, then messages within a batch have to be sorted by their publishing id.

The batch publisher with a batch limit (see Section 4.2) enables an application to batch messages from multiple asynchronous coroutines. Therefore, messages and their publishing ids can be enqueued out-of-order. The publisher sorts messages by their publishing id when sending messages to RabbitMQ Streams broker with *`flush()`* method.

The fast batch publisher (see Section 4.3) does not control the order of message publishing ids. An application takes the responsibility to provide them in a strictly increasing order.

# 8 Filter Messages

Applications can use RbFly with RabbitMQ Streams filtering feature to receive specific set of messages and reduce use of network bandwith.

The RabbitMQ Streams filtering feature needs to be used when producing and consuming messages with the broker.

The feature has been introduced in RabbitMQ 3.13, and RbFly supports it since version 0.9.0.

See also

- https://www.rabbitmq.com/blog/2023/10/16/stream-filtering
- https://www.rabbitmq.com/blog/2023/10/24/stream-filtering-internals

## 8.1 Publish and Read with Filtering

RabbitMQ Streams broker uses Bloom filter, a probabilistic data strucutre, to filter chunks of messages sent to a client. This has multiple implications

- an application sending messages to a stream has to provide filter values to the broker, so it can build Bloom filter structure server-side
- an application receiving messages from a stream has to check for invalid messages
  - a Bloom filter can match false-positive results, this is broker can send chunks with messages having non-requested filter values
  - chunks can contain mix of messages with both requested and non-requested filter values

An application needs to define a function to extract a filter value from a message. Filter value is always a string, for example if a message is a dictionary like:

```
{'amount': 10.0, 'country': 'country-code'}
```

then the function can be defined as:

```
def fv_extract(msg: MessageCtx) -> str:
    return msg.body['country']
```

Finally, use the function when creating stream publisher:

```
async with client.publisher('red-carrot', filter_extract=fv_extract) as publisher:
    await publisher.send({'amount': 100.0, 'country': 'jp'})
    await publisher.send({'amount': 200.0, 'country': 'ie'})
```

To receive messages for a country, subscribe to a stream with a message filter. Message filter consists of filter extract function and filter values, for example:

```
msg_filter = MessageFilter(fv_extract, {'ie'})
async for msg in client.subscribe('red-carrot', filter=msg_filter):
    print(msg)  # application receives messages with country `ie` only
```

## 8.2 Filtering Considerations

An application, when using RbFly library, orchestrates publishing of batches of messages to a RabbitMQ stream. This might give an illusion that an application can control how RabbitMQ Streams broker stores chunks of messages in segment files, and how the chunks are sent to a client. However, only RabbitMQ Streams broker decides when and how to create the chunks.

**Note**

There is some correlation between published batches of messages, and received chunks of messages. It is *not* guaranteed 1-to-1 match.

Additionally, a Bloom filter can point the broker to invalid chunks, which means that stream filtering cannot be fully efficient. Despite filtering, an application might receive non-requested data.

An example of RabbitMQ Streams broker splitting a batch of messages into two chunks is illustrated on Figure 8.1.

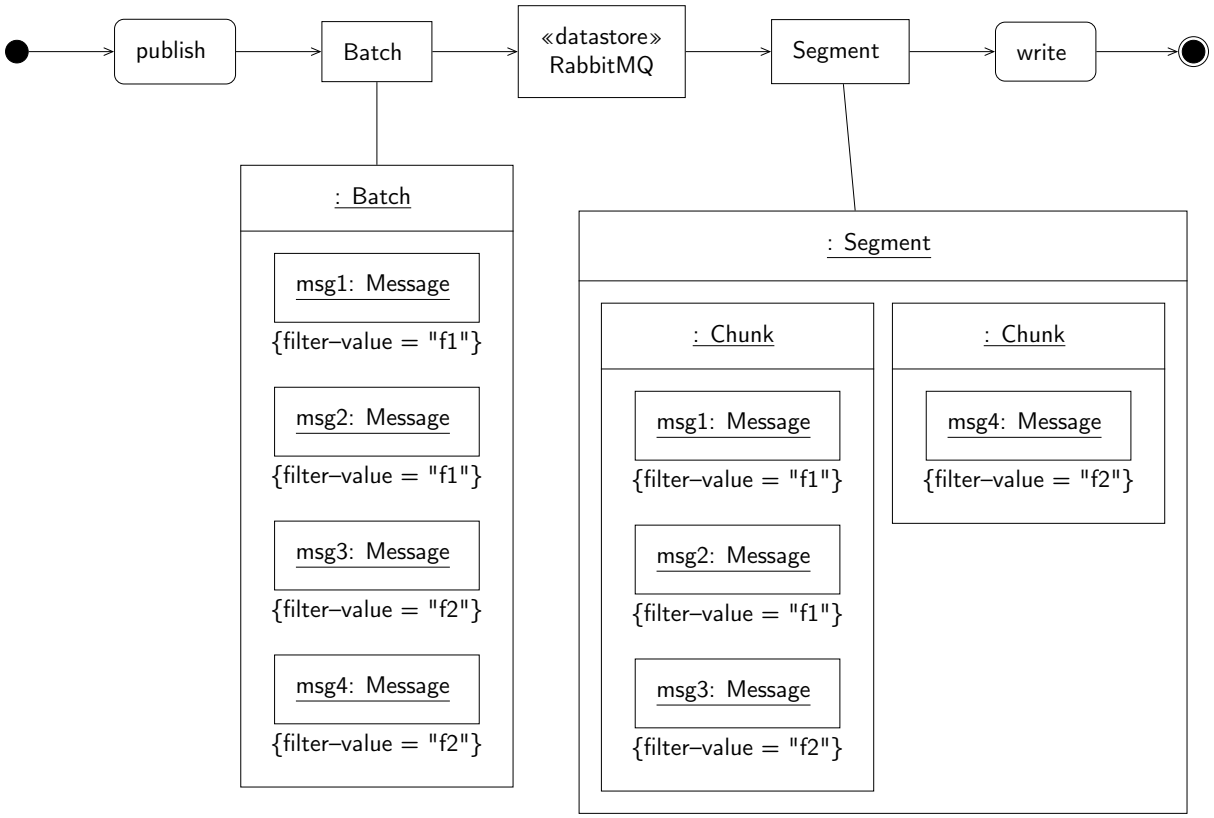


Figure 8.1: An example of RabbitMQ Streams broker splitting batch of messages into two chunks
An application publishes a batch of messages with a Bloom filter value assigned (two messages with *f1* value, and two messages with *f2* value). RabbitMQ receives the batch, splits messages into two chunks, and writes them as a segment file on a disk. This is not always observed, but very possible situation.

# 9 Manage Streams

There are multiple ways to manage RabbitMQ streams. The recommended way to create or delete a RabbitMQ stream is to use RabbitMQ CLI tools or infrastructure automation software.

## 9.1 RabbitMQ CLI Tools

Use rabbitmqadmin and rabbitmqctl command-line RabbitMQ tools to create and delete streams.

To create a RabbitMQ stream, declare a queue with *queue_type* argument set to *stream*:

```
$ rabbitmqadmin declare queue name=yellow.carrot queue_type=stream \
    arguments='{"x-max-age": "24h"}'
```

To delete a stream use *rabbitmqctl* command-line tool:

```
$ rabbitmqctl delete_queue yellow.carrot
```

## 9.2 Automation Tools

RabbitMQ streams can be created using automation tools like Terraform or Ansible

- Ansible RabbitMQ Queue Module
- Terraform RabbitMQ Queue Resource

As with RabbitMQ CLI tools, declare a RabbitMQ queue, and use *x-queue-type* argument set to *stream*. For example, for Ansible:

```
- name: Create a RabbitMQ stream
  become: yes
  community.rabbitmq.rabbitmq_queue:
    name: yellow.carrot
    arguments:
      x-queue-type: stream
      x-max-age: 1D
```

## 9.3 RbFly API

**Note**

This method of creating and deleting RabbitMQ streams is not recommended. It is used for tesing and in short examples only.

RbFly allows to create and delete streams programmatically with *create_stream()* and *delete_stream()* methods:

```
await client.create_stream('yellow.carrot')
await client.delete_stream('yellow.carrot')
```

# 10 Concurrency and Parallelism

RbFly uses asynchronous concurrency model, and avoids using threads.

Concurrency means an application executes and switches between multiple tasks on a single CPU core, while an application using threads might run a program on multiple processors of a machine. Threads can run code in a true, parallel manner. However, they are a limited resource, and should be managed by an application.

Performance of programs written in Python language, which try to use threads for parallelism, is hampered by Global Interpreter Lock (GIL). It blocks threads from utilizing multiple processors of a machine.

Despite the limitations of asynchronous concurrency model or Python threads, an application can use RbFly and threads to execute CPU bound tasks. This is possible when part of an application is implemented as a C extension. Such extensions can release GIL, which enables parallelism of a program.

## 10.1 Asynchronous Concurrency

The implementation of RbFly library is based on asynchronous concurrency model. In Python, asyncio framework provides an asynchronous event loop to execute tasks of RbFly library and of an application. They are executed in a single thread of the application.

After task is started, it initiaties an asynchronous operation, and yields control to the event loop allowing execution of other tasks. The event loop resumes execution of the task, when results of the asynchronous operation are available.

Concurrent execution of tasks can be blocked by a CPU bound task of an application. Such task does not yield control to an event loop, and prevents other tasks from running. For example, a CPU bound task might block RbFly from sending heartbeats by RabbitMQ Streams broker. The broker will close active connections if it does not receive heartbeats from an application in a timely manner. This problem can be avoided with threads.

## 10.2 Threads

Applications could run CPU bound tasks, if RbFly used threads to execute its tasks like sending heartbeats to RabbitMQ Streams broker.

However, RbFly avoids using threads. They bring additional cost like context switching and increased memory consumption, therefore threads should be managed like other limited resources. The use and management of threads is left up to an application.

A common pattern to maintain threads is to use a thread pool. This, in turn, limits concurrency and parallelism of an application, because the number of parallel tasks is limited by the size of a thread pool (Figure 10.1 and Figure 10.2). Finally, in Python language, the limitation is even more pronounced by GIL.

However, Python's GIL enables easy integration of libraries written in C language. Such libraries might release GIL when starting calculations, and enable parallelism of an application.

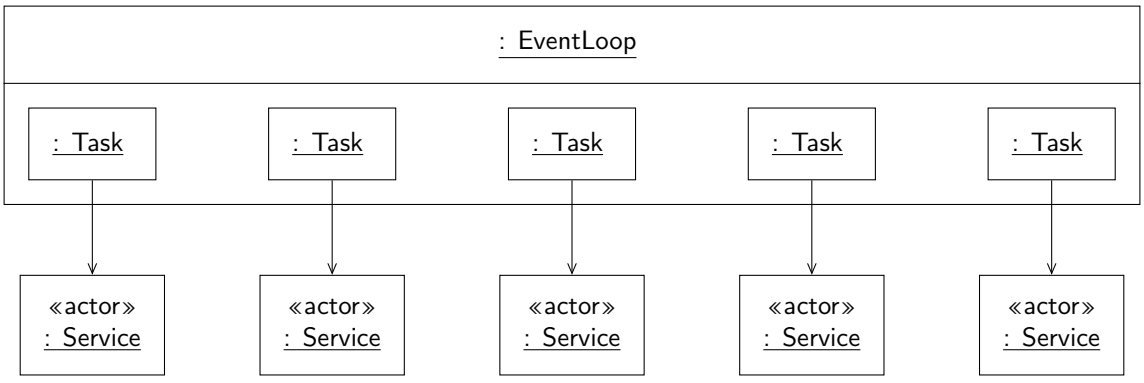


Figure 10.1: Concurrency of an event loop
An asynchronous event loop can run large number of tasks at once. An application can easily connect to all services, and execute required operations at the same time.

## 10.3 Example of Parallelism

NumPy is an example of a Python library, which might release GIL when executing calculations.

The demo presented in this subsection combines RbFly library with NumPy to demonstrate parallel execution of tasks.

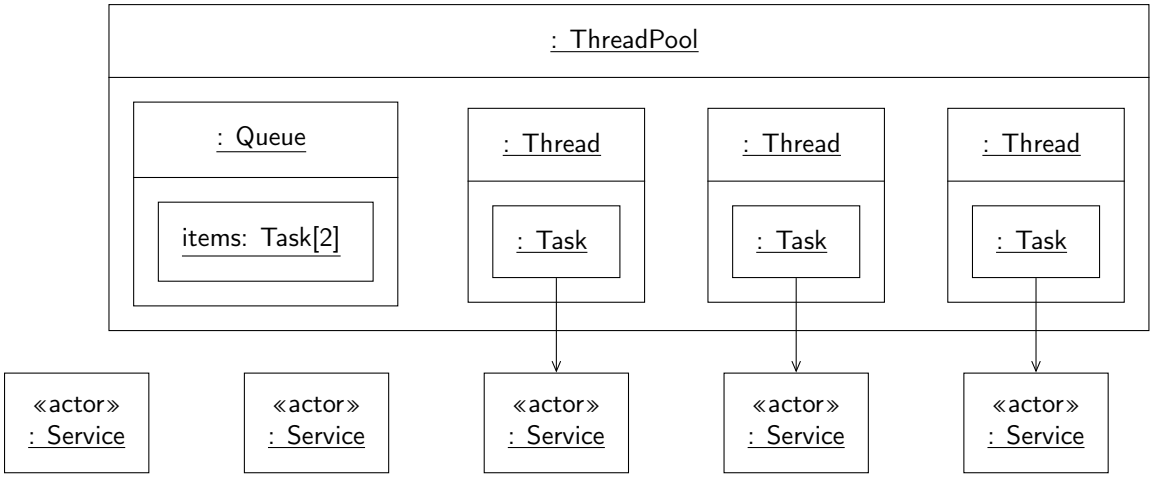


Figure 10.2: Concurrency of a thread pool
A thread pool can run limited number of tasks at once. Above, the thread pool has three workers. An application can connect to three services at the same time, and other two tasks are waiting in the thread pool queue.

The script consumes messages from a stream, and performs matrix multiplication with NumPy. The calculation lasts few minutes, and is executed in asyncio thread executor. During execution of the calculations in threads, RbFly is able to execute its tasks on asynchronous event loop

- heartbeats are sent RabbitMQ Streams broker
- additional task slowly consumes messages from the stream

**Note**

When script is run on a faster CPU, then increase size of a matrix with *-n* parameter of the script. A single matrix multiplication should run for 3 minutes, at least. This is to have overlap between the calculation and sending heartbeats to RabbitMQ Streams broker.

32 GB of RAM is required to run the script with the default parameters.

Alternatively, the script can be run with *-s* parameter. This starts NumPy calculations in the same thread, which is used by tasks of RbFly library. The calculations block RbFly from sending heartbeats to RabbitMQ Streams broker, and the broker eventually closes the current connection. Once application yields control to the event loop, then RbFly attempts reconnection to the server.

```
import asyncio
import logging
import numpy as np
from datetime import datetime
from functools import partial

import rbfly.streams as rbs

STREAM = 'rbfly-demo-stream-cpubound'

# send data to the stream
async def send_data(client: rbs.StreamsClient) -> None:
    async with client.publisher(STREAM) as publisher:
        for _ in range(3600):
            await publisher.send('hello')

# an additional task receiving data from the stream during calculations
```

(continues on next page)

(continued from previous page)

```
async def receive_data(client: rbs.StreamsClient) -> None:
    num_msg = 0
    print('{} start receiving messages'.format(datetime.now()))
    offset = rbs.Offset.FIRST
    async for msg in client.subscribe(STREAM, offset=offset):
        num_msg += 1
        await asyncio.sleep(1)
        if num_msg % 60 == 0:
            print('{} done receiving messages, num={}'.format(
                datetime.now(), num_msg
            ))

# read messages from the stream and perform CPU bound calculations
async def read_and_calculate(
        client: rbs.StreamsClient,
        use_threads: bool,
        size: int
) -> None:
    loop = asyncio.get_event_loop()
    offset = rbs.Offset.FIRST
    async for msg in client.subscribe(STREAM, offset=offset):
        if use_threads:
            await loop.run_in_executor(None, calculate, size)
        else:
            calculate(size)
        await asyncio.sleep(1)

def calculate(size: int):
    print('{} starting calculation'.format(datetime.now()))
    matrix = np.random.rand(size, size)
    result = matrix @ matrix @ matrix
    print('{} done calculating'.format(datetime.now()))

@rbs.connection
async def demo(client: rbs.StreamsClient, use_threads: bool, size: int) -> None:
    try:
        await client.create_stream(STREAM)
        await send_data(client)
        await asyncio.gather(
            read_and_calculate(client, use_threads, size),
            receive_data(client)
        )
    finally:
        await client.delete_stream(STREAM)

parser = argparse.ArgumentParser()
parser.add_argument(
    '-s', '--no-threads', action='store_true', default=False,
    help='use threads to perform cpu bound calculation'
)
parser.add_argument(
    '-n', '--size', default=24 * 1024, type=int,
    help='size of matrix used for calculation'
)
args = parser.parse_args()
```

(continues on next page)

(continued from previous page)

```
logging.basicConfig(level=logging.INFO)
client = rbs.streams_client('rabbitmq-stream://guest:guest@localhost')
use_threads = not args.no_threads

asyncio.run(demo(client, use_threads, args.size))
```

Part II

# RbFly Library Reference

# 11 RabbitMQ Streams API

## 11.1 Broker Connection API

| | |
|---|---|
| *rbfly.streams.streams_client* | Create RabbitMQ Streams client using connection URI. |
| *rbfly.streams.connection* | Decorator to manage RabbitMQ Streams client connection. |
| *rbfly.streams.StreamsClient* | RabbitMQ Streams client. |

rbfly.streams.**streams_client**(*uri: str*, */*, *ssl: SSLContext | None = None*) → *StreamsClient*

Create RabbitMQ Streams client using connection URI.

**Parameters**

- **uri** – Connection URI.
- **ssl** – TLS/SSL context object.

rbfly.streams.**connection**(*coro: Callable[[Concatenate[Trc, P]], Coroutine[Any, Any, T]]*) → Callable[[Concatenate[Trc, P]], Coroutine[Any, Any, T]]

rbfly.streams.**connection**(*coro: Callable[[Concatenate[Trc, P]], AsyncIterator[T]]*) → Callable[[Concatenate[Trc, P]], AsyncIterator[T]]

Decorator to manage RabbitMQ Streams client connection.

Streams client implements connection manager abstract class.

Streams client has to be the first parameter of coroutine *coro*.

Streams client is disconnected on exit of the coroutine using connection manager API.

**Parameters**

**coro** – Coroutine using RabbitMQ Streams client.

**class** rbfly.streams.**StreamsClient**(*connection_info: ConnectionInfo*)

RabbitMQ Streams client.

**async create_stream**(*stream: str*) → None

Create RabbitMQ stream.

Method ignores error received from RabbitMQ Streams broker if the stream exists.

**Parameters**

**stream** – RabbitMQ stream name.

**async delete_stream**(*stream: str*) → None

Delete RabbitMQ stream.

Method ignores error received from RabbitMQ Streams broker if the stream does not exist.

**Parameters**

**stream** – RabbitMQ stream name.

**publisher**(*stream: str*, *, *name: str | None = None*, *filter_extract: BloomFilterExtract | None = None*) → tp.AsyncContextManager[*Publisher*]

**publisher**(*stream: str*, *, *name: str | None = None*, *filter_extract: BloomFilterExtract | None = None*, *cls: type[T]*) → tp.AsyncContextManager[T]

Create publisher for RabbitMQ stream.

The single message, AMQP publisher is used by default.

The stream must exist.

Publisher reference name is used for deduplication of messages. By default, the publisher name is <hostname>/<pid>. Override it, if this scheme does not work for a specific application, i.e. when using threads.

**Parameters**

- **stream** – RabbitMQ stream name.
- **name** – RabbitMQ stream publisher reference name.
- **filter_extract** – Function to extract values for stream Bloom filter.
- **cls** – Publisher class.

**See also**

- *rbfly.streams.types.BloomFilterExtract*
- *rbfly.streams.Publisher*
- *rbfly.streams.PublisherBatchFast*
- *rbfly.streams.PublisherBatchLimit*

**async subscribe**(*stream: str*, *, *offset:* Offset = *Offset.NEXT*, *filter:* MessageFilter | *None* = *None*, *timeout: float = 0*, *amqp: bool = True*) → AsyncIterator[rbfly.streams.types.AMQPBody]

Subscribe to the stream and iterate over messages.

**Parameters**

- **stream** – Name of RabbitMQ stream to subscribe to.
- **offset** – RabbitMQ Streams offset specification.
- **filter** – RabbitMQ stream message filter.
- **timeout** – Raise timeout error if no message received within specified time (in seconds).
- **amqp** – Messages are in AMQP 1.0 format if true. Otherwise no AMQP decoding.

**See also**

- *rbfly.streams.MessageFilter*
- *rbfly.streams.types.BloomFilterExtract*

**async write_offset**(*stream: str*, *reference: str*, *value: int* | *None* = *None*) → None

Write RabbitMQ stream offset value using the reference string.

When offset value is not specified, then the last message context is retrieved and its stream offset value is stored. If there is no last message context to retrieve, then method does nothing.

**Parameters**

- **stream** – Name of RabbitMQ stream.
- **reference** – Offset reference string.
- **value** – Offset value to be stored.

## 11.2 Publisher and Subscriber API

| | |
|---|---|
| *rbfly.streams.stream_message_ctx* | Create message context for RabbitMQ Streams publisher. |
| *rbfly.streams.get_message_ctx* | Get current context of AMQP message. |
| *rbfly.streams.StreamsClient* | RabbitMQ Streams client. |
| *rbfly.streams.Publisher* | RabbitMQ Streams publisher for sending a single message. |
| *rbfly.streams.PublisherBatchLimit* | RabbitMQ Streams publisher for sending limited batch of messages. |
| *rbfly.streams.PublisherBatchFast* | RabbitMQ Streams publisher for sending a batch of messages. |
| *rbfly.streams.MessageCtx* | AMQP message context. |
| *rbfly.streams.Offset* | Offset specification for RabbitMQ stream subscription. |
| *rbfly.streams.MessageFilter* | Message filter for RabbitMQ Streams subscription. |
| *rbfly.streams.types.BloomFilterExtract* | Function to extract values from messages for stream Bloom filter. |

rbfly.streams.**stream_message_ctx**(*body: AMQPBody*, *, *publish_id: int | None = None*, *app_properties: AMQPAppProperties = {}*) → *MessageCtx*

Create message context for RabbitMQ Streams publisher.

Message publish id is optional - a publisher assigns one if not specified. Message publish id can be used for message deduplication. If an application provides message publish ids, then it is its responsibility to track them and keep the ids strictly increasing.

Application properties are part of AMQP message. The properties can be used for filtering or routing.

**Parameters**

- **body** – Message data to be sent to a stream.
- **publish_id** – Message publish id.
- **app_properties** – Application properties, part of AMQP message.

rbfly.streams.**get_message_ctx**() → *MessageCtx*

Get current context of AMQP message.

**class** rbfly.streams.**Publisher**

RabbitMQ Streams publisher for sending a single message.

**See also**

- *rbfly.streams.PublisherBatchLimit*
- *rbfly.streams.PublisherBatchFast*

**name**

name: unicode Publisher reference name.

**stream**

stream: unicode RabbitMQ stream name.

**message_id**

message_id: int Last value of published message id.

**async send**(*self*, *message: AMQPBody | MessageCtx*) → None

Send AMQP message to a RabbitMQ stream.

The asynchronous coroutine waits for message delivery confirmation from RabbitMQ Streams broker.

A *message* is simply application data of type `AMQPBody`, or message context (class *MessageCtx*).

**Parameters**

**message** – AMQP message to publish.

**See also**

- *stream_message_ctx()*
- AMQPBody
- *MessageCtx*

class rbfly.streams.**PublisherBatchLimit**

RabbitMQ Streams publisher for sending limited batch of messages.

The publisher performs coordination between the batch and flush asynchronous coroutines to allow sending only limited number of messages.

**See also**

- *rbfly.streams.PublisherBatchFast*
- *rbfly.streams.Publisher*

**name**

name: unicode Publisher reference name.

**stream**

stream: unicode RabbitMQ stream name.

**message_id**

message_id: int Last value of published message id.

**async batch**(*self*, *message: AMQPBody | MessageCtx*, *, *max_len: int*) → None

Enqueue AMQP message for batch processing with RabbitMQ Streams broker.

The asynchronous coroutine blocks when *max_len* messages are enqueued. To unblock, call *PublisherBatchLimit.flush()* method.

A *message* is simply application data of type `AMQPBody`, or message context (class *MessageCtx*).

**Parameters**

- **message** – AMQP message to publish.
- **max_len** – Maximum number of messages in a batch.

**See also**

- *PublisherBatchLimit.flush()*
- *stream_message_ctx()*
- AMQPBody
- *MessageCtx*

**async flush**(*self*) → None

Flush all enqueued messages and unblock *PublisherBatchLimit.batch()* asynchronous coroutines.

**See also**

*PublisherBatchLimit.batch()*

**class** rbfly.streams.**PublisherBatchFast**

RabbitMQ Streams publisher for sending a batch of messages.

The number of messages in a single batch is limited by the maximum length of the Python list type on a given platform.

- *rbfly.streams.PublisherBatchLimit*
- *rbfly.streams.Publisher*

**name**

name: unicode Publisher reference name.

**stream**

stream: unicode RabbitMQ stream name.

**message_id**

message_id: int Last value of published message id.

**batch**(*self*, *message: AMQPBody | MessageCtx*) → None

Enqueue AMQP message for batch processing with RabbitMQ Streams broker.

A *message* is simply application data of type `AMQPBody`, or message context (class *MessageCtx*).

**Parameters**
**message** – AMQP message to publish.

**See also**

- *PublisherBatchFast.flush()*
- *stream_message_ctx()*
- `AMQPBody`
- *MessageCtx*

**async flush**(*self*) → None

Flush all enqueued messages.

**class** rbfly.streams.**MessageCtx**

AMQP message context.

**Variables**

- **body** – Message body.
- **header** – Message header.
- **delivery_annotations** – Message delivery annotations.
- **annotations** – Message annotations.
- **properties** – Message properties.
- **app_properties** – Application properties.

- **footer** – Message footer.
- **stream_offset** – RabbitMQ stream offset value.
- **stream_timestamp** – RabbitMQ stream offset timestamp value.
- **stream_publish_id** – RabbitMQ stream message publishing id.

**class** rbfly.streams.**Offset**(*type: OffsetType*, *value: int | float | str | None = None*)

Offset specification for RabbitMQ stream subscription.

**FIRST: *Offset* = Offset.FIRST**

Receive all messages, starting with the very first message in a stream. Equivalent to *Offset.offset(0)*.

**LAST: *Offset* = Offset.LAST**

Receive messages from a streams starting with first message stored in the current stream chunk.

**NEXT: *Offset* = Offset.NEXT**

Receive new messages from a stream only. Default offset specification.

**static offset**(*offset: int*) → *Offset*

Create offset specification with offset value.

**Parameters**
**offset** – Offset value.

**static reference**(*reference: str*) → *Offset*

Create offset specification, which queries and stores stream offset with offset reference.

**Parameters**
**reference** – Offset reference string.

**static timestamp**(*timestamp: float*) → *Offset*

Create offset specification with timestamp value.

**Parameters**
**timestamp** – Unix timestamp in seconds since epoch.

**class** rbfly.streams.**MessageFilter**(*extract: Callable[[*MessageCtx*], str]*, *values: set[str]*)

Message filter for RabbitMQ Streams subscription.

RabbitMQ Streams broker uses filter values to filter chunks of messages with Bloom filter.

Extract function is used to remove messages with non-requested filter values, i.e. false positives of a Bloom filter.

**Variables**

- **extract** – Function to extract values for Bloom Filter.
- **values** – Set of values to filter stream messages with.

rbfly.streams.types.**BloomFilterExtract**

Function to extract values from messages for stream Bloom filter.

alias of Callable[[*MessageCtx*], str]

## 11.3 Data Types

| | |
|---|---|
| *rbfly.types.Symbol* | Symbolic value from a constrained domain as defined by AMQP 1.0. |
| *rbfly.types.AMQPScalar* | AMQP simple type. |
| *rbfly.types.AMQPBody* | Application data sent as AMQP message. |
| *rbfly.types.AMQPAnnotations* | AMQP message annotations. |
| *rbfly.types.AMQPAppProperties* | Application properties sent with AMQP message. |

**class** rbfly.types.**Symbol**(*name: str*)

Symbolic value from a constrained domain as defined by AMQP 1.0.

The class is also exported via `rbfly.streams` module.

rbfly.types.**AMQPScalar**: *TypeAlias* = **None | str | bool | int | float | datetime.datetime | uuid.UUID | rbfly.types.Symbol | bytes**

AMQP simple type.

The type is also exported via `rbfly.streams` module.

rbfly.types.**AMQPBody**: *TypeAlias* = **collections.abc.Sequence['AMQPBody'] | dict['AMQPBody', 'AMQPBody'] | None | str | bool | int | float | datetime.datetime | uuid.UUID | rbfly.types.Symbol | bytes**

Application data sent as AMQP message.

It is sent by a publisher and received by a subscriber.

The type is also exported via `rbfly.streams` module.

rbfly.types.**AMQPAnnotations**

AMQP message annotations.

The type is also exported via `rbfly.streams` module.

alias of `dict`[*Symbol* | `int`, `Sequence`[AMQPBody] | `dict`[AMQPBody, AMQPBody] | `None` | `str` | `bool` | `int` | `float` | `datetime` | `UUID` | *Symbol* | `bytes`]

rbfly.types.**AMQPAppProperties**

Application properties sent with AMQP message.

The type is also exported via `rbfly.streams` module.

alias of `dict`[`str`, `None` | `str` | `bool` | `int` | `float` | `datetime` | `UUID` | *Symbol* | `bytes`]

## 11.4 Deprecated API

Deprecated since version 0.7.0:

**class** rbfly.streams.**PublisherBatch**

Use *rbfly.streams.PublisherBatchFast* class instead.

**class** rbfly.streams.**PublisherBatchMem**

Use *rbfly.streams.PublisherBatchLimit* class instead.

# 12 Connection URI

The string specifying RabbitMQ Streams connection is URI in the format:

```
rabbitmq-stream://[user[:password]@][host[:port][/vhost]]
```

or for TLS:

```
rabbitmq-stream+tls://[user[:password]@][host[:port][/vhost]]
```

The description of the URI parts is as follows

**user**
Name of an user connecting to RabbitMQ Streams broker.

**password**
Password of the user.

**host**
RabbitMQ Streams broker hostname, *localhost* by default.

**port**
RabbitMQ Streams broker port, *5552* by default.

**vhost**
Virtual host of RabbitMQ Streams broker, it is / (slash) by default.

Examples of valid URI values:

```
rabbitmq-stream://
rabbitmq-stream://localhost
rabbitmq-stream://localhost/
rabbitmq-stream://good-rabbit:1552/carrot
rabbitmq-stream://user@good-rabbit
rabbitmq-stream://user:passta@good-rabbit/carrot
```

Examples of valid URI values for TLS connections:

```
rabbitmq-stream+tls://
rabbitmq-stream+tls://localhost
rabbitmq-stream+tls://localhost/
rabbitmq-stream+tls://good-rabbit:1551/carrot
rabbitmq-stream+tls://user@good-rabbit
rabbitmq-stream+tls://user:passta@good-rabbit/carrot
```

# 13 Message Data Types

**Note**

Support for AMQP 1.0 encoding and decoding is work in progress.

Message body, sent to a RabbitMQ stream, is any Python object having one of the types

`bytes`
A data section of AMQP message containing opaque binary data. It can carry any binary message encoded with third-party encoders like MessagePack.

The data type is also accepted as a member of a container type like sequence, or dictionary.

`None`
An AMQP message with null value.

`str`
An AMQP message with string value encoded with Unicode.

`bool`
An AMQP message with boolean value (true or false).

`int`
An AMQP message with integer value. The value is encoded with AMQP type as defined by the table

| Value Range | AMQP Type Name |
|---|---|
| $[-2^{31}, 2^{31} - 1]$ | int |
| $[-2^{63}, 2^{63} - 1]$ | long |
| $[2^{63}, 2^{64} - 1]$ | ulong |

All other, AMQP integer types are parsed as Python `int` type.

`float`
: An AMQP message with value of AMQP type double. AMQP type float is parsed as value of Python type float, as well.

**sequence**
: An AMQP message with value of AMQP type list. Python `list` and `tuple` types are supported when serializing to AMQP format. Always decoded into a Python list.

`dict`
: An AMQP message with value of AMQP type map. The order of key-value pairs in a map is preserved on encoding and decoding. In AMQP standard, the order of key-value pairs is semantically important, but this is not the case for Python dictionary.

`datetime.datetime`
: An AMQP message with value of AMQP type timestamp. Unaware datetime object is encoded as UTC timestamp value. Decoded datetime object has UTC timezone.

`uuid.UUID`
: An AMQP message with an universally unique identifier as defined by RFC-4122, section 4.1.2.

`Symbol`
: Symbolic value from a constrained domain. The value is an ASCII string.

# 14 Changelog

## 14.1 Year 2024

### 2024-10-17, ver. 0.10.0

- improve amqp message decoding
- fix bugs related to rabbitmq broker automatic reconnection
- rabbitmq 3.13 or later is required
- python 3.11 or later is required

### 2024-07-11, ver. 0.9.0

- add support for stream message filtering
- improve reporting of errors of rabbitmq streams protocol
- add documentation topic about application concurrency

### 2024-05-21, ver. 0.8.4

- remove use of deprecated python functions

### 2024-05-04, ver. 0.8.3

- fix errors in *rbfly.streams.connection* decorator

### 2024-03-19, ver. 0.8.2

- improve type annotations for *rbfly.streams.connection* decorator

### 2024-01-07, ver. 0.8.1

- fix typo: *stream_messsage_ctx* -> *stream_message_ctx*

## 14.2 Year 2023

### 2023-11-02, ver. 0.8.0

- allow tls connections to rabbitmq streams broker
- publish performance test results for amqp 1.0 codecs
- flush batch publisher on exit from an asynchronous context manager of a publisher
- avoid publisher ids and subscriber ids overflow by reusing and controlling the ids

### 2023-09-15, ver. 0.7.3

- add support for null values in amqp messages
- use cython 3.0.2 to build rbfly binary extensions

### 2023-07-09, ver. 0.7.2

- fix binary only decoder of rabbitmq streams messages
- fix reading duplicate messages when re-subscribing to a rabbitmq stream

### 2023-06-25, ver. 0.7.1

- fix reconnection to rabbitmq streams broker, which was failing due to hidden subscriber class attributes

### 2023-06-21, ver. 0.7.0

- rbfly is beta software, now
- add support for deduplication of published messages
- add support for publishing messages with amqp application properties
- publish performance test results for message publishing
- include aiokafka and apache kafka in message publishing performance tests
- fix potential buffer overflow issues when decoding message publishing confirmations
- use cython 3.0.0b2 to build rbfly binary extensions
- rename `PublisherBatch` class as `PublisherBatchFast` class
- rename `PublisherBatchMem` class as `PublisherBatchLimit` class

### 2023-05-12, ver. 0.6.2

- fix parsing of virtual host in rabbitmq streams uri, i.e. */vhost* -> *vhost* but / remains as /

### 2023-03-18, ver. 0.6.1

- fix message id value type; it is 64-bit, not 32-bit value

### 2023-02-04, ver. 0.6.0

- fix various buffer overflow issues
- split batch of messages, and send multiple batches, when the batch is too big to fit into rabbitmq streams protocol frame
- increase initial credit value to 128; this improves performance of reading messages with rabbitmq 3.11.8; the possible memory footprint of 400 MB to read messages still seems to be acceptable
- adapt to rabbitmq 3.11.8 changes related to the handling of stream subscription credit

## 14.3 Year 2022

### 2022-12-22, ver. 0.5.6

- fix processing of credit value of rabbitmq stream subscription, when messages were published in batches

### 2022-12-19, ver. 0.5.5

- improve maintenance of rabbitmq stream's subscription credit

### 2022-11-23, ver. 0.5.4

- fix authentication with rabbitmq streams broker (send password if specified, not username)

### 2022-11-11, ver. 0.5.3

- use recursive definition for container amqp types, i.e. sequence can contain another sequence or a dictionary
- documentation improvements

### 2022-11-01, ver. 0.5.2

- fix encoding of binary data in containers, i.e. list of elements having bytes type

### 2022-10-16, ver. 0.5.1

- fix deadlock of rabbitmq streams protocol publish method when no messages are sent

### 2022-10-09, ver. 0.5.0

- implement rabbitmq streams batching publisher with memory protection and concurrently working from multiple asynchronous coroutines
- send heartbeat if there is no communication from rabbitmq streams broker and restart connection if no response

### 2022-09-12, ver. 0.4.2

- allow sending messages via a streams publisher from multiple coroutines
- improve rabbitmq streams connection decorator type annotations

### 2022-08-16, ver. 0.4.1

- minor fixes post 0.4.0 release

### 2022-08-16, ver. 0.4.0

- add support for encoding and decoding of timestamp and UUID values in amqp 1.0 codec
- add support for decoding of message annotations and application properties of amqp message
- fix amqp decoder when dealing with unsigned char values (i.e. size of a byte string)
- fix storing of rabbitmq streams timestamp in message context object

### 2022-05-23, ver. 0.3.0

- api change: do not store stream offset value in stream subscription method to give more control over the action to a programmer
- implement rabbitmq streams client method to store stream offset value
- implement function to get message context for a received stream message

### 2022-05-13, ver. 0.2.0

- change publisher reference name scheme to `<hostname>/<pid>`
- allow to set publisher reference name when creating a stream publisher
- improve handling of rabbitmq streams disconnections
- increase sleep time when retrying stream subscription or rabbitmq streams broker reconnection (current sleep time is too aggressive with simple broker restart)

### 2022-04-13, ver. 0.1.1

- fix handling of multiple disconnection requests, i.e. when multiple coroutines share a client
- properly propagate asynchronous generators via the connection decorator

### 2022-04-06, ver. 0.1.0

- improvements to decoding of amqp messages
- fix compilation issues on macos and freebsd operating systems

### 2022-03-25, ver. 0.0.4

- add support for value of type list in amqp messages

### 2022-03-24, ver. 0.0.3

- add support for values of type boolean, integer, float, and dictionaries in amqp messages

### 2022-01-24, ver. 0.0.2

- fix encoding of messages

### 2022-01-16, ver. 0.0.1

- initial release
- genindex

# Index