

---

# **Kombu Documentation**

***Release 3.0.37***

**Ask Solem**

November 30, 2016



<b>1</b>	<b>kombu - Messaging library for Python</b>	<b>3</b>
1.1	Features . . . . .	3
1.2	Transport Comparison . . . . .	4
1.3	Installation . . . . .	5
1.4	Terminology . . . . .	6
1.5	Getting Help . . . . .	7
1.6	Bug tracker . . . . .	7
1.7	Contributing . . . . .	7
1.8	License . . . . .	7
<b>2</b>	<b>User Guide</b>	<b>9</b>
2.1	Introduction . . . . .	9
2.2	Connections and transports . . . . .	10
2.3	Producers . . . . .	13
2.4	Consumers . . . . .	15
2.5	Examples . . . . .	19
2.6	Simple Interface . . . . .	21
2.7	Connection and Producer Pools . . . . .	23
2.8	Serialization . . . . .	25
<b>3</b>	<b>Frequently Asked Questions</b>	<b>29</b>
3.1	Questions . . . . .	29
<b>4</b>	<b>API Reference</b>	<b>31</b>
4.1	Connection . . . . .	31
4.2	Exchange . . . . .	38
4.3	Queue . . . . .	40
4.4	Message Producer . . . . .	44
4.5	Message Consumer . . . . .	45
4.6	Common Utilities - kombu.common . . . . .	48
4.7	Mixin Classes - kombu.mixins . . . . .	50
4.8	kombu.simple . . . . .	52
4.9	Clocks and Synchronization - kombu.clocks . . . . .	53
4.10	kombu.compat . . . . .	55
4.11	kombu.pidbox . . . . .	61
4.12	kombu.exceptions . . . . .	63
4.13	Logging - kombu.log . . . . .	63
4.14	kombu.connection . . . . .	64

4.15	Message Objects - kombu.message	71
4.16	kombu.compression	72
4.17	General Pools - kombu.pools	73
4.18	kombu.abstract	75
4.19	Async Utilities - kombu.syn	76
4.20	Event Loop - kombu.async	76
4.21	Event Loop Implementation - kombu.async.hub	77
4.22	Semaphores - kombu.async.semaphore	78
4.23	Timer - kombu.async.timer	79
4.24	Debugging Utils - kombu.async.debug	80
4.25	kombu.transport	81
4.26	kombu.transport.pyamqp	81
4.27	kombu.transport.qpid	100
4.28	Authentication	100
4.29	Transport Options	100
4.30	kombu.transport.memory	133
4.31	kombu.transport.redis	134
4.32	kombu.transport.zmq	138
4.33	kombu.transport.beanstalk	138
4.34	kombu.transport.mongodb	139
4.35	kombu.transport.couchdb	140
4.36	kombu.transport.zookeeper	141
4.37	kombu.transport.filesystem	142
4.38	kombu.transport.django	143
4.39	Django Models - kombu.transport.django.models	144
4.40	Django Managers - kombu.transport.django.managers	145
4.41	Django Management - clean_kombu_messages	145
4.42	kombu.transport.sqlalchemy	146
4.43	kombu.transport.SLMQ	146
4.44	kombu.transport.pyro	147
4.45	kombu.transport.amqplib	148
4.46	kombu.transport.base	150
4.47	kombu.transport.virtual	151
4.48	kombu.transport.virtual.exchange	156
4.49	kombu.transport.virtual.scheduling	157
4.50	kombu.serialization	157
4.51	Utilities - kombu.utils	159
4.52	Evented I/O - kombu.utils.eventio	161
4.53	Rate limiting - kombu.utils.limits	161
4.54	Compat. utilities - kombu.utils.compat	162
4.55	Debugging - kombu.utils.debug	162
4.56	String Encoding - kombu.utils.encoding	162
4.57	kombu.utils.functional	163
4.58	kombu.utils.url	163
4.59	Text utilities - kombu.utils.text	164
4.60	Generic RabbitMQ manager - kombu.utils.amq_manager	164
4.61	Python2 to Python3 utilities - kombu.five	164
<b>5</b>	<b>Change history</b>	<b>165</b>
5.1	3.0.37	165
5.2	3.0.36	165
5.3	3.0.35	165
5.4	3.0.34	166
5.5	3.0.33	166

5.6	3.0.32	166
5.7	3.0.31	166
5.8	3.0.30	167
5.9	3.0.29	167
5.10	3.0.28	167
5.11	3.0.27	168
5.12	3.0.26	168
5.13	3.0.25	169
5.14	3.0.24	169
5.15	3.0.23	170
5.16	3.0.22	170
5.17	3.0.21	171
5.18	3.0.20	171
5.19	3.0.19	171
5.20	3.0.18	171
5.21	3.0.17	171
5.22	3.0.16	172
5.23	3.0.15	172
5.24	3.0.14	173
5.25	3.0.13	173
5.26	3.0.12	174
5.27	3.0.11	174
5.28	3.0.10	175
5.29	3.0.9	175
5.30	3.0.8	176
5.31	3.0.7	176
5.32	3.0.6	176
5.33	3.0.5	177
5.34	3.0.4	177
5.35	3.0.3	177
5.36	3.0.2	178
5.37	3.0.1	178
5.38	3.0.0	178
5.39	2.5.16	181
5.40	2.5.15	181
5.41	2.5.14	181
5.42	2.5.13	181
5.43	2.5.12	182
5.44	2.5.11	182
5.45	2.5.10	183
5.46	2.5.9	183
5.47	2.5.8	184
5.48	2.5.7	184
5.49	2.5.6	185
5.50	2.5.5	185
5.51	2.5.4	185
5.52	2.5.3	186
5.53	2.5.2	186
5.54	2.5.2	186
5.55	2.5.1	186
5.56	2.5.0	186
5.57	2.4.10	188
5.58	2.4.9	188
5.59	2.4.8	188

5.60	2.4.7	189
5.61	2.4.6	189
5.62	2.4.5	190
5.63	2.4.4	190
5.64	2.4.3	190
5.65	2.4.2	190
5.66	2.4.1	190
5.67	2.4.0	191
5.68	2.3.2	191
5.69	2.3.1	191
5.70	2.3.0	191
5.71	2.2.6	192
5.72	2.2.5	193
5.73	2.2.4	193
5.74	2.2.3	193
5.75	2.2.2	194
5.76	2.2.1	194
5.77	2.2.0	195
5.78	2.1.8	198
5.79	2.1.7	198
5.80	2.1.6	198
5.81	2.1.5	198
5.82	2.1.4	199
5.83	2.1.3	199
5.84	2.1.2	200
5.85	2.1.1	200
5.86	2.1.0	200
5.87	2.0.0	201
5.88	1.5.1	202
5.89	1.5.0	202
5.90	1.4.3	203
5.91	1.4.2	203
5.92	1.4.1	204
5.93	1.4.0	204
5.94	1.3.5	205
5.95	1.3.4	205
5.96	1.3.3	205
5.97	1.3.2	205
5.98	1.3.1	206
5.99	1.3.0	206
5.100	1.2.1	208
5.101	1.2.0	208
5.102	1.1.6	209
5.103	1.1.5	209
5.104	1.1.4	209
5.105	1.1.3	210
5.106	1.1.2	210
5.107	1.1.1	210
5.108	1.1.0	210
5.109	1.0.7	211
5.110	1.0.6	212
5.111	1.0.5	213
5.112	1.0.4	213
5.113	1.0.3	213

5.114 1.0.2	213
5.115 1.0.1	214
5.116 1.0.0	214
5.117 0.1.0	214
<b>6 Indices and tables</b>	<b>215</b>
<b>Python Module Index</b>	<b>217</b>





Contents:



---

## kombu - Messaging library for Python

---

**Version** 3.0.37

*Kombu* is a messaging library for Python.

The aim of *Kombu* is to make messaging in Python as easy as possible by providing an idiomatic high-level interface for the AMQ protocol, and also provide proven and tested solutions to common messaging problems.

**AMQP** is the Advanced Message Queuing Protocol, an open standard protocol for message orientation, queuing, routing, reliability and security, for which the **RabbitMQ** messaging server is the most popular implementation.

### 1.1 Features

- Allows application authors to support several message server solutions by using pluggable transports.
  - AMQP transport using the `py-amqp`, `librabbitmq`, or `qpuid-python` client libraries.
  - High performance AMQP transport written in C - when using `librabbitmq`

This is automatically enabled if `librabbitmq` is installed:

```
$ pip install librabbitmq
```

- Virtual transports makes it really easy to add support for non-AMQP transports. There is already built-in support for `Redis`, `Beanstalk`, `Amazon SQS`, `CouchDB`, `MongoDB`, `ZeroMQ`, `ZooKeeper`, `SoftLayer MQ` and `Pyro`.
- You can also use the `SQLAlchemy` and `Django ORM` transports to use a database as the broker.
- In-memory transport for unit testing.
- Supports automatic encoding, serialization and compression of message payloads.
- Consistent exception handling across transports.
- The ability to ensure that an operation is performed by gracefully handling connection and channel errors.
- Several annoyances with `amqplib` has been fixed, like supporting timeouts and the ability to wait for events on more than one channel.
- Projects already using `carrot` can easily be ported by using a compatibility layer.

For an introduction to AMQP you should read the article [Rabbits and warrens](#), and the [Wikipedia article about AMQP](#).

## 1.2 Transport Comparison

Client	Type	Direct	Topic	Fanout
<i>amqp</i>	Native	Yes	Yes	Yes
<i>qpq</i>	Native	Yes	Yes	Yes
<i>redis</i>	Virtual	Yes	Yes	Yes (PUB/SUB)
<i>mongodb</i>	Virtual	Yes	Yes	Yes
<i>beanstalk</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>SQS</i>	Virtual	Yes	Yes <sup>1</sup>	Yes <sup>2</sup>
<i>couchdb</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>zookeeper</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>in-memory</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>django</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>sqlalchemy</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>SLMQ</i>	Virtual	Yes	Yes <sup>1</sup>	No

### 1.2.1 Documentation

Kombu is using Sphinx, and the latest documentation can be found here:

<https://kombu.readthedocs.io/>

### 1.2.2 Quick overview

```

from kombu import Connection, Exchange, Queue

media_exchange = Exchange('media', 'direct', durable=True)
video_queue = Queue('video', exchange=media_exchange, routing_key='video')

def process_media(body, message):
    print body
    message.ack()

# connections
with Connection('amqp://guest:guest@localhost//') as conn:

    # produce
    producer = conn.Producer(serializer='json')
    producer.publish({'name': '/tmp/lolcat1.avi', 'size': 1301013},
                    exchange=media_exchange, routing_key='video',
                    declare=[video_queue])

    # the declare above, makes sure the video queue is declared
    # so that the messages can be delivered.
    # It's a best practice in Kombu to have both publishers and
    # consumers declare the queue. You can also declare the
    # queue manually using:
    #     video_queue(conn).declare()

# consume

```

<sup>1</sup>Declarations only kept in memory, so exchanges/queues must be declared by all clients that needs them.

<sup>2</sup>Fanout supported via storing routing tables in SimpleDB. Disabled by default, but can be enabled by using the `supports_fanout` transport option.

```

with conn.Consumer(video_queue, callbacks=[process_media]) as consumer:
    # Process messages and handle events on all channels
    while True:
        conn.drain_events()

# Consume from several queues on the same channel:
video_queue = Queue('video', exchange=media_exchange, key='video')
image_queue = Queue('image', exchange=media_exchange, key='image')

with connection.Consumer([video_queue, image_queue],
                        callbacks=[process_media]) as consumer:
    while True:
        connection.drain_events()

```

Or handle channels manually:

```

with connection.channel() as channel:
    producer = Producer(channel, ...)
    consumer = Producer(channel)

```

All objects can be used outside of with statements too, just remember to close the objects after use:

```

from kombu import Connection, Consumer, Producer

connection = Connection()
# ...
connection.release()

consumer = Consumer(channel_or_connection, ...)
consumer.register_callback(my_callback)
consumer.consume()
# ....
consumer.cancel()

```

*Exchange* and *Queue* are simply declarations that can be pickled and used in configuration files etc.

They also support operations, but to do so they need to be bound to a channel.

Binding exchanges and queues to a connection will make it use that connections default channel.

```

>>> exchange = Exchange('tasks', 'direct')

>>> connection = Connection()
>>> bound_exchange = exchange(connection)
>>> bound_exchange.delete()

# the original exchange is not affected, and stays unbound.
>>> exchange.delete()
raise NotBoundError: Can't call delete on Exchange not bound to
a channel.

```

## 1.3 Installation

You can install *Kombu* either via the Python Package Index (PyPI) or from source.

To install using *pip*:

```
$ pip install kombu
```

To install using *easy\_install*:

```
$ easy_install kombu
```

If you have downloaded a source tarball you can install it by doing the following,:

```
$ python setup.py build
# python setup.py install # as root
```

## 1.4 Terminology

There are some concepts you should be familiar with before starting:

- Producers

Producers sends messages to an exchange.

- Exchanges

Messages are sent to exchanges. Exchanges are named and can be configured to use one of several routing algorithms. The exchange routes the messages to consumers by matching the routing key in the message with the routing key the consumer provides when binding to the exchange.

- Consumers

Consumers declares a queue, binds it to a exchange and receives messages from it.

- Queues

Queues receive messages sent to exchanges. The queues are declared by consumers.

- Routing keys

Every message has a routing key. The interpretation of the routing key depends on the exchange type. There are four default exchange types defined by the AMQP standard, and vendors can define custom types (so see your vendors manual for details).

These are the default exchange types defined by AMQP/0.8:

- Direct exchange

Matches if the routing key property of the message and the *routing\_key* attribute of the consumer are identical.

- Fan-out exchange

Always matches, even if the binding does not have a routing key.

- Topic exchange

Matches the routing key property of the message by a primitive pattern matching scheme. The message routing key then consists of words separated by dots (".", like domain names), and two special characters are available; star ("\*") and hash ("#"). The star matches any word, and the hash matches zero or more words. For example *\*.stock.#* matches the routing keys *usd.stock* and *eur.stock.db* but not *stock.nasdaq*.

## 1.5 Getting Help

### 1.5.1 Mailing list

Join the `carrot-users` mailing list.

## 1.6 Bug tracker

If you have any suggestions, bug reports or annoyances please report them to our issue tracker at <http://github.com/celery/kombu/issues/>

## 1.7 Contributing

Development of *Kombu* happens at Github: <http://github.com/celery/kombu>

You are highly encouraged to participate in the development. If you don't like Github (for some reason) you're welcome to send regular patches.

## 1.8 License

This software is licensed under the *New BSD License*. See the *LICENSE* file in the top distribution directory for the full license text.





**Release** 3.0

**Date** November 30, 2016

## 2.1 Introduction

### 2.1.1 What is messaging?

In times long ago people didn't have email. They had the postal service, which with great courage would deliver mail from hand to hand all over the globe. Soldiers deployed at wars far away could only communicate with their families through the postal service, and posting a letter would mean that the recipient wouldn't actually receive the letter until weeks or months, sometimes years later.

It's hard to imagine this today when people are expected to be available for phone calls every minute of the day.

So humans need to communicate with each other, this shouldn't be news to anyone, but why would applications?

One example is banks. When you transfer money from one bank to another, your bank sends a message to a central clearinghouse. The clearinghouse then records and coordinates the transaction. Banks need to send and receive millions and millions of messages every day, and losing a single message would mean either losing your money (bad) or the banks money (very bad)

Another example is the stock exchanges, which also have a need for very high message throughputs and have strict reliability requirements.

Email is a great way for people to communicate. It is much faster than using the postal service, but still using email as a means for programs to communicate would be like the soldier above, waiting for signs of life from his girlfriend back home.

### 2.1.2 Messaging Scenarios

- Request/Reply

The request/reply pattern works like the postal service example. A message is addressed to a single recipient, with a return address printed on the back. The recipient may or may not reply to the message by sending it back to the original sender.

Request-Reply is achieved using *direct* exchanges.

- Broadcast

In a broadcast scenario a message is sent to all parties. This could be none, one or many recipients.

Broadcast is achieved using *fanout* exchanges.

- Publish/Subscribe

In a publish/subscribe scenario producers publish messages to topics, and consumers subscribe to the topics they are interested in.

If no consumers subscribe to the topic, then the message will not be delivered to anyone. If several consumers subscribe to the topic, then the message will be delivered to all of them.

Pub-sub is achieved using *topic* exchanges.

### 2.1.3 Reliability

For some applications reliability is very important. Losing a message is a critical situation that must never happen. For other applications losing a message is fine, it can maybe recover in other ways, or the message is resent anyway as periodic updates.

AMQP defines two built-in delivery modes:

- persistent

Messages are written to disk and survives a broker restart.

- transient

Messages may or may not be written to disk, as the broker sees fit to optimize memory contents. The messages will not survive a broker restart.

Transient messaging is by far the fastest way to send and receive messages, so having persistent messages comes with a price, but for some applications this is a necessary cost.

## 2.2 Connections and transports

### 2.2.1 Basics

To send and receive messages you need a transport and a connection. There are several transports to choose from (amqp, librabbitmq, redis, qpid, in-memory, etc.), and you can even create your own. The default transport is amqp.

Create a connection using the default transport:

```
>>> from kombu import Connection
>>> connection = Connection('amqp://guest:guest@localhost:5672//')
```

The connection will not be established yet, as the connection is established when needed. If you want to explicitly establish the connection you have to call the `connect()` method:

```
>>> connection.connect()
```

You can also check whether the connection is connected:

```
>>> connection.connected
True
```

Connections must always be closed after use:

```
>>> connection.close()
```

But best practice is to release the connection instead, this will release the resource if the connection is associated with a connection pool, or close the connection if not, and makes it easier to do the transition to connection pools later:

```
>>> connection.release()
```

**See also:**

*Connection and Producer Pools*

Of course, the connection can be used as a context, and you are encouraged to do so as it makes it harder to forget releasing open resources:

```
with Connection() as connection:
    # work with connection
```

## 2.2.2 URLs

Connection parameters can be provided as an URL in the format:

```
transport://userid:password@hostname:port/virtual_host
```

All of these are valid URLs:

```
# Specifies using the amqp transport only, default values
# are taken from the keyword arguments.
amqp://

# Using Redis
redis://localhost:6379/

# Using Redis over a Unix socket
redis+socket:///tmp/redis.sock

# Using Qpid
qpid://localhost/

# Using virtual host '/foo'
amqp://localhost//foo

# Using virtual host 'foo'
amqp://localhost/foo
```

The query part of the URL can also be used to set options, e.g.:

```
amqp://localhost/myvhost?ssl=1
```

See *Keyword arguments* for a list of supported options.

A connection without options will use the default connection settings, which is using the localhost host, default port, user name *guest*, password *guest* and virtual host *"/*". A connection without arguments is the same as:

```
>>> Connection('amqp://guest:guest@localhost:5672//')
```

The default port is transport specific, for AMQP this is 5672.

Other fields may also have different meaning depending on the transport used. For example, the Redis transport uses the *virtual\_host* argument as the redis database number.

## 2.2.3 Keyword arguments

The `Connection` class supports additional keyword arguments, these are:

- hostname** Default host name if not provided in the URL.
- userid** Default user name if not provided in the URL.
- password** Default password if not provided in the URL.
- virtual\_host** Default virtual host if not provided in the URL.
- port** Default port if not provided in the URL.
- transport** Default transport if not provided in the URL. Can be a string specifying the path to the class. (e.g. `kombu.transport.pyamqp.Transport`), or one of the aliases: `pyamqp`, `librabbitmq`, `redis`, `qpido`, `memory`, and so on.
- ssl** Use SSL to connect to the server. Default is `False`. Only supported by the `amqp` and `qpido` transports.
- insist** Insist on connecting to a server. *No longer supported, relic from AMQP 0.8*
- connect\_timeout** Timeout in seconds for connecting to the server. May not be supported by the specified transport.
- transport\_options** A dict of additional connection arguments to pass to alternate kombu channel implementations. Consult the transport documentation for available options.

## 2.2.4 AMQP Transports

There are 4 transports available for AMQP use.

1. `pyamqp` uses the pure Python library `amqp`, automatically installed with Kombu.
2. `librabbitmq` uses the high performance transport written in C. This requires the `librabbitmq` Python package to be installed, which automatically compiles the C library.
3. `amqp` tries to use `librabbitmq` but falls back to `pyamqp`.
4. `qpido` uses the pure Python library `qpido.messaging`, automatically installed with Kombu. The Qpid library uses AMQP, but uses custom extensions specifically supported by the Apache Qpid Broker.

For the highest performance, you should install the `librabbitmq` package. To ensure `librabbitmq` is used, you can explicitly specify it in the transport URL, or use `amqp` to have the fallback.

## 2.2.5 Transport Comparison

Client	Type	Direct	Topic	Fanout
<i>amqp</i>	Native	Yes	Yes	Yes
<i>qp</i>	Native	Yes	Yes	Yes
<i>redis</i>	Virtual	Yes	Yes	Yes (PUB/SUB)
<i>mongodb</i>	Virtual	Yes	Yes	Yes
<i>beanstalk</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>SQS</i>	Virtual	Yes	Yes <sup>1</sup>	Yes <sup>2</sup>
<i>couchdb</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>zookeeper</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>in-memory</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>django</i>	Virtual	Yes	Yes <sup>1</sup>	No
<i>sqlalchemy</i>	Virtual	Yes	Yes <sup>1</sup>	No

## 2.3 Producers

### 2.3.1 Basics

### 2.3.2 Serialization

See *Serialization*.

### 2.3.3 Reference

```
class kombu.Producer(channel, exchange=None, routing_key=None, serializer=None,
                    auto_declare=None, compression=None, on_return=None)
```

Message Producer.

#### Parameters

- **channel** – Connection or channel.
- **exchange** – Optional default exchange.
- **routing\_key** – Optional default routing key.
- **serializer** – Default serializer. Default is “*json*”.
- **compression** – Default compression method. Default is no compression.
- **auto\_declare** – Automatically declare the default exchange at instantiation. Default is `True`.
- **on\_return** – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to `publish()` is used. This callback needs the following signature: (*exception*, *exchange*, *routing\_key*, *message*). Note that the producer needs to drain events to use this feature.

#### **auto\_declare = True**

By default the exchange is declared at instantiation. If you want to declare manually then you can set this to `False`.

<sup>1</sup>Declarations only kept in memory, so exchanges/queues must be declared by all clients that needs them.

<sup>2</sup>Fanout supported via storing routing tables in SimpleDB. Disabled by default, but can be enabled by using the `supports_fanout` transport option.

**compression = None**

Default compression method. Disabled by default.

**declare ()**

Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

**exchange = None**

Default exchange

**maybe\_declare (entity, retry=False, \*\*retry\_policy)**

Declare the exchange if it hasn't already been declared during this session.

**on\_return = None**

Basic return callback.

**publish (body, routing\_key=None, delivery\_mode=None, mandatory=False, immediate=False, priority=0, content\_type=None, content\_encoding=None, serializer=None, headers=None, compression=None, exchange=None, retry=False, retry\_policy=None, declare=[], expiration=None, \*\*properties)**

Publish message to the specified exchange.

**Parameters**

- **body** – Message body.
- **routing\_key** – Message routing key.
- **delivery\_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content\_type** – Content type. Default is auto-detect.
- **content\_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry\_policy** – Retry configuration, this is the keywords supported by `ensure()`.
- **expiration** – A TTL in seconds can be specified per message. Default is no expiration.
- **\*\*properties** – Additional message properties, see AMQP spec.

**revive (channel)**

Revive the producer after connection loss.

**routing\_key = ''**

Default routing key.

**serializer = None**

Default serializer to use. Default is JSON.

## 2.4 Consumers

### 2.4.1 Basics

The `Consumer` takes a connection (or channel) and a list of queues to consume from. Several consumers can be mixed to consume from different channels, as they all bind to the same connection, and `drain_events` will drain events from all channels on that connection.

**Note:** Kombu since 3.0 will only accept json/binary or text messages by default, to allow deserialization of other formats you have to specify them in the `accept` argument:

```
Consumer(conn, accept=['json', 'pickle', 'msgpack', 'yaml'])
```

Draining events from a single consumer:

```
with Consumer(connection, queues, accept=['json']):
    connection.drain_events(timeout=1)
```

Draining events from several consumers:

```
from kombu.utils import nested

with connection.channel(), connection.channel() as (channel1, channel2):
    with nested(Consumer(channel1, queues1, accept=['json']),
                Consumer(channel2, queues2, accept=['json'])):
        connection.drain_events(timeout=1)
```

Or using `ConsumerMixin`:

```
from kombu.mixins import ConsumerMixin

class C(ConsumerMixin):

    def __init__(self, connection):
        self.connection = connection

    def get_consumers(self, Consumer, channel):
        return [
            Consumer(queues, callbacks=[self.on_message], accept=['json']),
        ]

    def on_message(self, body, message):
        print("RECEIVED MESSAGE: %r" % (body, ))
        message.ack()

C(connection).run()
```

and with multiple channels again:

```
from kombu import Consumer
from kombu.mixins import ConsumerMixin

class C(ConsumerMixin):
    channel2 = None

    def __init__(self, connection):
```

```

self.connection = connection

def get_consumers(self, _, default_channel):
    self.channel2 = default_channel.connection.channel()
    return [Consumer(default_channel, queues1,
                    callbacks=[self.on_message],
                    accept=['json']),
            Consumer(self.channel2, queues2,
                    callbacks=[self.on_special_message],
                    accept=['json'])]

def on_consumer_end(self, connection, default_channel):
    if self.channel2:
        self.channel2.close()

C(connection).run()

```

## 2.4.2 Reference

**class** kombu.Consumer(*channel*, *queues=None*, *no\_ack=None*, *auto\_declare=None*, *callbacks=None*, *on\_decode\_error=None*, *on\_message=None*, *accept=None*, *tag\_prefix=None*)  
 Message consumer.

### Parameters

- **channel** – see channel.
- **queues** – see queues.
- **no\_ack** – see no\_ack.
- **auto\_declare** – see auto\_declare
- **callbacks** – see callbacks.
- **on\_message** – See on\_message
- **on\_decode\_error** – see on\_decode\_error.

### exception ContentDisallowed

Consumer does not allow this content-type.

### Consumer.accept = None

List of accepted content-types.

An exception will be raised if the consumer receives a message with an untrusted content type. By default all content-types are accepted, but not if `kombu.disable_untrusted_serializers()` was called, in which case only json is allowed.

### Consumer.add\_queue(*queue*)

Add a queue to the list of queues to consume from.

This will not start consuming from the queue, for that you will have to call `consume()` after.

### Consumer.add\_queue\_from\_dict(*queue*, *\*\*options*)

This method is deprecated.

Instead please use:

```
consumer.add_queue(Queue.from_dict(d))
```



**Consumer.auto\_declare = True**

By default all entities will be declared at instantiation, if you want to handle this manually you can set this to `False`.

**Consumer.callbacks = None**

List of callbacks called in order when a message is received.

The signature of the callbacks must take two arguments: (*body*, *message*), which is the decoded message body and the *Message* instance (a subclass of *Message*).

**Consumer.cancel ()**

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

**Consumer.cancel\_by\_queue (queue)**

Cancel consumer by queue name.

**Consumer.channel = None**

The connection/channel to use for this consumer.

**Consumer.close ()**

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

**Consumer.consume (no\_ack=None)**

Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues ( use `cancel_by_queue ()` ).

**Parameters no\_ack** – See `no_ack`.

**Consumer.consuming\_from (queue)**

Return `True` if the consumer is currently consuming from queue'.

**Consumer.declare ()**

Declare queues, exchanges and bindings.

This is done automatically at instantiation if `auto_declare` is set.

**Consumer.flow (active)**

Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

**Consumer.no\_ack = None**

Flag for automatic message acknowledgment. If enabled the messages are automatically acknowledged by the broker. This can increase performance but means that you have no control of when the message is removed.

Disabled by default.

**Consumer.on\_decode\_error = None**

Callback called when a message can't be decoded.

The signature of the callback must take two arguments: (*message*, *exc*), which is the message that can't be decoded and the exception that occurred while trying to decode it.

`Consumer.on_message = None`

Optional function called whenever a message is received.

When defined this function will be called instead of the `receive()` method, and callbacks will be disabled.

So this can be used as an alternative to `callbacks` when you don't want the body to be automatically decoded. Note that the message will still be decompressed if the message has the `compression` header set.

The signature of the callback must take a single argument, which is the raw message object (a subclass of *Message*).

Also note that the `message.body` attribute, which is the raw contents of the message body, may in some cases be a read-only `buffer` object.

`Consumer.purge()`

Purge messages from all queues.

**Warning:** This will *delete all ready messages*, there is no undo operation.

`Consumer.qos(prefetch_size=0, prefetch_count=0, apply_global=False)`

Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the `no_ack` option is set.

#### Parameters

- **prefetch\_size** – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning “no specific limit”, although other prefetch limits may still apply.
- **prefetch\_count** – Specify the prefetch window in terms of whole messages.
- **apply\_global** – Apply new settings globally on all channels.

`Consumer.queues = None`

A single *Queue*, or a list of queues to consume from.

`Consumer.receive(body, message)`

Method called when a message is received.

This dispatches to the registered `callbacks`.

#### Parameters

- **body** – The decoded message body.
- **message** – The *Message* instance.

**Raises `NotImplementedError`** – If no consumer callbacks have been registered.

`Consumer.recover(rewrite=False)`

Redeliver unacknowledged messages.

Asks the broker to redeliver all unacknowledged messages on the specified channel.

**Parameters `requeue`** – By default the messages will be redelivered to the original recipient. With `requeue` set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

`Consumer.register_callback` (*callback*)

Register a new callback to be called when a message is received.

The signature of the callback needs to accept two arguments: (*body*, *message*), which is the decoded message body and the *Message* instance (a subclass of *Message*).

`Consumer.revive` (*channel*)

Revive consumer after connection loss.

## 2.5 Examples

### 2.5.1 Hello World Example

Below example uses *Simple Interface* to send helloworld message through message broker (rabbitmq) and print received message

hello\_publisher.py:

```
from kombu import Connection
import datetime

with Connection('amqp://guest:guest@localhost:5672//') as conn:
    simple_queue = conn.SimpleQueue('simple_queue')
    message = 'helloworld, sent at %s' % datetime.datetime.today()
    simple_queue.put(message)
    print('Sent: %s' % message)
    simple_queue.close()
```

hello\_consumer.py:

```
from kombu import Connection

with Connection('amqp://guest:guest@localhost:5672//') as conn:
    simple_queue = conn.SimpleQueue('simple_queue')
    message = simple_queue.get(block=True, timeout=1)
    print("Received: %s" % message.payload)
    message.ack()
    simple_queue.close()
```

### 2.5.2 Task Queue Example

Very simple task queue using pickle, with primitive support for priorities using different queues.

queues.py:

```
from kombu import Exchange, Queue

task_exchange = Exchange('tasks', type='direct')
task_queues = [Queue('hipri', task_exchange, routing_key='hipri'),
               Queue('midpri', task_exchange, routing_key='midpri'),
               Queue('lopri', task_exchange, routing_key='lopri')]
```

worker.py:

```
from kombu.mixins import ConsumerMixin
from kombu.log import get_logger
from kombu.utils import kwdict, reprcall

from .queues import task_queues

logger = get_logger(__name__)

class Worker(ConsumerMixin):

    def __init__(self, connection):
        self.connection = connection

    def get_consumers(self, Consumer, channel):
        return [Consumer(queues=task_queues,
                        accept=['pickle', 'json'],
                        callbacks=[self.process_task])]

    def process_task(self, body, message):
        fun = body['fun']
        args = body['args']
        kwargs = body['kwargs']
        logger.info('Got task: %s', reprcall(fun.__name__, args, kwargs))
        try:
            fun(*args, **kwdict(kwargs))
        except Exception as exc:
            logger.error('task raised exception: %r', exc)
        message.ack()

if __name__ == '__main__':
    from kombu import Connection
    from kombu.utils.debug import setup_logging
    # setup root logger
    setup_logging(loglevel='INFO', loggers=[])

    with Connection('amqp://guest:guest@localhost:5672//') as conn:
        try:
            worker = Worker(conn)
            worker.run()
        except KeyboardInterrupt:
            print('bye bye')
```

tasks.py:

```
def hello_task(who="world"):
    print("Hello %s" % (who, ))
```

client.py:

```
from kombu.pools import producers

from .queues import task_exchange

priority_to_routing_key = {'high': 'hipri',
```

```

        'mid': 'midpri',
        'low': 'lopri'})

def send_as_task(connection, fun, args=(), kwargs={}, priority='mid'):
    payload = {'fun': fun, 'args': args, 'kwargs': kwargs}
    routing_key = priority_to_routing_key[priority]

    with producers[connection].acquire(block=True) as producer:
        producer.publish(payload,
                        serializer='pickle',
                        compression='bzip2',
                        exchange=task_exchange,
                        declare=[task_exchange],
                        routing_key=routing_key)

if __name__ == '__main__':
    from kombu import Connection
    from .tasks import hello_task

    connection = Connection('amqp://guest:guest@localhost:5672//')
    send_as_task(connection, fun=hello_task, args=('Kombu', ), kwargs={},
                 priority='high')

```

## 2.6 Simple Interface

- *Sending and receiving messages*

`kombu.simple` is a simple interface to AMQP queuing. It is only slightly different from the `Queue` class in the Python Standard Library, which makes it excellent for users with basic messaging needs.

Instead of defining exchanges and queues, the simple classes only requires two arguments, a connection channel and a name. The name is used as the queue, exchange and routing key. If the need arises, you can specify a `Queue` as the name argument instead.

In addition, the `Connection` comes with shortcuts to create simple queues using the current connection:

```

>>> queue = connection.SimpleQueue('myqueue')
>>> # ... do something with queue
>>> queue.close()

```

This is equivalent to:

```

>>> from kombu import SimpleQueue, SimpleBuffer

>>> channel = connection.channel()
>>> queue = SimpleBuffer(channel)
>>> # ... do something with queue
>>> channel.close()
>>> queue.close()

```

## 2.6.1 Sending and receiving messages

The simple interface defines two classes; *SimpleQueue*, and *SimpleBuffer*. The former is used for persistent messages, and the latter is used for transient, buffer-like queues. They both have the same interface, so you can use them interchangeably.

Here is an example using the *SimpleQueue* class to produce and consume logging messages:

```
import socket
import datetime
from time import time
from kombu import Connection

class Logger(object):

    def __init__(self, connection, queue_name='log_queue',
                 serializer='json', compression=None):
        self.queue = connection.SimpleQueue(queue_name)
        self.serializer = serializer
        self.compression = compression

    def log(self, message, level='INFO', context={}):
        self.queue.put({'message': message,
                       'level': level,
                       'context': context,
                       'hostname': socket.gethostname(),
                       'timestamp': time()},
                      serializer=self.serializer,
                      compression=self.compression)

    def process(self, callback, n=1, timeout=1):
        for i in xrange(n):
            log_message = self.queue.get(block=True, timeout=1)
            entry = log_message.payload # deserialized data.
            callback(entry)
            log_message.ack() # remove message from queue

    def close(self):
        self.queue.close()

if __name__ == '__main__':
    from contextlib import closing

    with Connection('amqp://guest:guest@localhost:5672//') as conn:
        with closing(Logger(conn)) as logger:

            # Send message
            logger.log('Error happened while encoding video',
                     level='ERROR',
                     context={'filename': 'cutekitten.mpg'})

            # Consume and process message

            # This is the callback called when a log message is
            # received.
            def dump_entry(entry):
                date = datetime.datetime.fromtimestamp(entry['timestamp'])
```

```

print(['%s %s %s] %s %r' % (date,
                             entry['hostname'],
                             entry['level'],
                             entry['message'],
                             entry['context']))

# Process a single message using the callback above.
logger.process(dump_entry, n=1)

```

## 2.7 Connection and Producer Pools

### 2.7.1 Default Pools

Kombu ships with two global pools: one connection pool, and one producer pool.

These are convenient and the fact that they are global may not be an issue as connections should often be limited at the process level, rather than per thread/application and so on, but if you need custom pools per thread see *Custom Pool Groups*.

#### The connection pool group

The connection pools are available as `kombu.pools.connections`. This is a pool group, which means you give it a connection instance, and you get a pool instance back. We have one pool per connection instance to support multiple connections in the same app. All connection instances with the same connection parameters will get the same pool:

```

>>> from kombu import Connection
>>> from kombu.pools import connections

>>> connections[Connection('redis://localhost:6379')]
<kombu.connection.ConnectionPool object at 0x101805650>
>>> connections[Connection('redis://localhost:6379')]
<kombu.connection.ConnectionPool object at 0x101805650>

```

Let's acquire and release a connection:

```

from kombu import Connection
from kombu.pools import connections

connection = Connection('redis://localhost:6379')

with connections[connection].acquire(block=True) as conn:
    print('Got connection: %r' % (connection.as_uri(), ))

```

**Note:** The `block=True` here means that the acquire call will block until a connection is available in the pool. Note that this will block forever in case there is a deadlock in your code where a connection is not released. There is a `timeout` argument you can use to safeguard against this (see `kombu.connection.Resource.acquire()`).

If blocking is disabled and there aren't any connections left in the pool an `kombu.exceptions.ConnectionLimitExceeded` exception will be raised.

That's about it. If you need to connect to multiple brokers at once you can do that too:

```
from kombu import Connection
from kombu.pools import connections

c1 = Connection('amqp://')
c2 = Connection('redis://')

with connections[c1].acquire(block=True) as conn1:
    with connections[c2].acquire(block=True) as conn2:
        # ....
```

## 2.7.2 The producer pool group

This is a pool group just like the connections, except that it manages *Producer* instances used to publish messages.

Here is an example using the producer pool to publish a message to the news exchange:

```
from kombu import Connection, Exchange
from kombu.pools import producers

# The exchange we send our news articles to.
news_exchange = Exchange('news')

# The article we want to send
article = {'title': 'No cellular coverage on the tube for 2012',
          'ingress': 'yadda yadda yadda'}

# The broker where our exchange is.
connection = Connection('amqp://guest:guest@localhost:5672/')

with producers[connection].acquire(block=True) as producer:
    producer.publish(
        article,
        exchange=new_exchange,
        routing_key='domestic',
        declare=[news_exchange],
        serializer='json',
        compression='zlib')
```

### Setting pool limits

By default every connection instance has a limit of 200 connections. You can change this limit using `kombu.pools.set_limit()`. You are able to grow the pool at runtime, but you can't shrink it, so it is best to set the limit as early as possible after your application starts:

```
>>> from kombu import pools
>>> pools.set_limit()
```

### Resetting all pools

You can close all active connections and reset all pool groups by using the `kombu.pools.reset()` function. Note that this will not respect anything currently using these connections, so will just drag the connections away from under their feet: you should be very careful before you use this.

Kombu will reset the pools if the process is forked, so that forked processes start with clean pool groups.



## 2.7.3 Custom Pool Groups

To maintain your own pool groups you should create your own `Connections` and `kombu.pools.Producers` instances:

```
from kombu import pools
from kombu import Connection

connections = pools.Connections(limit=100)
producers = pools.Producers(limit=connections.limit)

connection = Connection('amqp://guest:guest@localhost:5672//')

with connections[connection].acquire(block=True):
    # ...
```

If you want to use the global limit that can be set with `set_limit()` you can use a special value as the `limit` argument:

```
from kombu import pools

connections = pools.Connections(limit=pools.use_default_limit)
```

## 2.8 Serialization

### 2.8.1 Serializers

By default every message is encoded using `JSON`, so sending Python data structures like dictionaries and lists works. `YAML`, `msgpack` and Python's built-in `pickle` module is also supported, and if needed you can register any custom serialization scheme you want to use.

By default Kombu will only load `JSON` messages, so if you want to use other serialization format you must explicitly enable them in your consumer by using the `accept` argument:

```
Consumer(conn, [queue], accept=['json', 'pickle', 'msgpack'])
```

The `accept` argument can also include MIME-types.

Each option has its advantages and disadvantages.

***json*** – **JSON is supported in many programming languages, is now** a standard part of Python (since 2.6), and is fairly fast to decode using the modern Python libraries such as *cjson* or *simplejson*.

The primary disadvantage to *JSON* is that it limits you to the following data types: strings, Unicode, floats, boolean, dictionaries, and lists. Decimals and dates are notably missing.

Also, binary data will be transferred using Base64 encoding, which will cause the transferred data to be around 34% larger than an encoding which supports native binary types.

However, if your data fits inside the above constraints and you need cross-language support, the default setting of *JSON* is probably your best choice.

***pickle*** – **If you have no desire to support any language other than** Python, then using the *pickle* encoding will gain you the support of all built-in Python data types (except class instances), smaller messages when sending binary files, and a slight speedup over *JSON* processing.

---

### Pickle and Security

The pickle format is very convenient as it can serialize and deserialize almost any object, but this is also a concern for security.

Carefully crafted pickle payloads can do almost anything a regular Python program can do, so if you let your consumer automatically decode pickled objects you must make sure to limit access to the broker so that untrusted parties do not have the ability to send messages!

By default Kombu uses pickle protocol 2, but this can be changed using the `PICKLE_PROTOCOL` environment variable or by changing the global `kombu.serialization.pickle_protocol` flag.

**yaml** – **YAML has many of the same characteristics as json**, except that it natively supports more data types (including dates, recursive references, etc.)

However, the Python libraries for YAML are a good bit slower than the libraries for JSON.

If you need a more expressive set of data types and need to maintain cross-language compatibility, then *YAML* may be a better fit than the above.

To instruct *Kombu* to use an alternate serialization method, use one of the following options.

1. Set the serialization option on a per-producer basis:

```
>>> producer = Producer(channel,
...                       exchange=exchange,
...                       serializer="yaml")
```

2. Set the serialization option per message:

```
>>> producer.publish(message, routing_key=rkey,
...                   serializer="pickle")
```

Note that a *Consumer* do not need the serialization method specified. They can auto-detect the serialization method as the content-type is sent as a message header.

### 2.8.2 Sending raw data without Serialization

In some cases, you don't need your message data to be serialized. If you pass in a plain string or Unicode object as your message and a custom *content\_type*, then *Kombu* will not waste cycles serializing/deserializing the data.

You can optionally specify a *content\_encoding* for the raw data:

```
>>> with open("~/my_picture.jpg", "rb") as fh:
...     producer.publish(fh.read(),
...                       content_type="image/jpeg",
...                       content_encoding="binary",
...                       routing_key=rkey)
```

The *Message* object returned by the *Consumer* class will have a *content\_type* and *content\_encoding* attribute.

### 2.8.3 Creating extensions using Setuptools entry-points

A package can also register new serializers using Setuptools entry-points.

The entry-point must provide the name of the serializer along with the path to a tuple providing the rest of the args: *encoder\_function*, *decoder\_function*, *content\_type*, *content\_encoding*.

An example entrypoint could be:

```
from setuptools import setup

setup(
    entry_points={
        'kombu.serializers': [
            'my_serializer = my_module.serializer:register_args'
        ]
    }
)
```

Then the module `my_module.serializer` would look like:

```
register_args = (my_encoder, my_decoder, 'application/x-mimetype', 'utf-8')
```

When this package is installed the new ‘my\_serializer’ serializer will be supported by Kombu.

---

### Buffer Objects

The decoder function of custom serializer must support both strings and Python’s old-style buffer objects.

Python pickle and json modules usually don’t do this via its `loads` function, but you can easily add support by making a wrapper around the `load` function that takes file objects instead of strings.

Here’s an example wrapping `pickle.loads()` in such a way:

```
import pickle
from kombu.serialization import BytesIO, register

def loads(s):
    return pickle.load(BytesIO(s))

register('my_pickle', pickle.dumps, loads,
        content_type='application/x-pickle2',
        content_encoding='binary')
```



---

## Frequently Asked Questions

---

### 3.1 Questions

#### 3.1.1 Q: `Message.reject` doesn't work?

**Answer:** Earlier versions of RabbitMQ did not implement `basic.reject`, so make sure your version is recent enough to support it.

#### 3.1.2 Q: `Message.requeue` doesn't work?

**Answer:** See `Message.reject` doesn't work?



## API Reference

**Release** 3.0**Date** November 30, 2016

- *Connection*
- *Exchange*
- *Queue*
- *Message Producer*
- *Message Consumer*

Messaging library for Python

`kombu.enable_insecure_serializers` (*choices*=['pickle', 'yaml', 'msgpack'])

Enable serializers that are considered to be unsafe.

Will enable `pickle`, `yaml` and `msgpack` by default, but you can also specify a list of serializers (by name or content type) to enable.

`kombu.disable_insecure_serializers` (*allowed*=['json'])

Disable untrusted serializers.

Will disable all serializers except `json` or you can specify a list of deserializers to allow.

---

**Note:** Producers will still be able to serialize data in these formats, but consumers will not accept incoming data using the untrusted content types.

---

## 4.1 Connection

```
class kombu.Connection (hostname='localhost', userid=None, password=None, virtual_host=None,
                        port=None, insist=False, ssl=False, transport=None, connect_timeout=5,
                        transport_options=None, login_method=None, uri_prefix=None, heartbeat=0,
                        failover_strategy='round-robin', alternates=None, **kwargs)
```

A connection to the broker.

**Parameters** **URL** – Broker URL, or a list of URLs, e.g.

```
Connection('amqp://guest:guest@localhost:5672//')
Connection('amqp://foo;amqp://bar', failover_strategy='round-robin')
```

```
Connection('redis://', transport_options={
    'visibility_timeout': 3000,
})

import ssl
Connection('amqp://', login_method='EXTERNAL', ssl={
    'ca_certs': '/etc/pki/tls/certs/something.crt',
    'keyfile': '/etc/something/system.key',
    'certfile': '/etc/something/system.cert',
    'cert_reqs': ssl.CERT_REQUIRED,
})
```

---

### SSL compatibility

SSL currently only works with the py-amqp, amqplib, and qpid transports. For other transports you can use stunnel.

---

#### Parameters

- **ssl** – Use SSL to connect to the server. Default is `False`. May not be supported by the specified transport.
- **transport** – Default transport if not specified in the URL.
- **connect\_timeout** – Timeout in seconds for connecting to the server. May not be supported by the specified transport.
- **transport\_options** – A dict of additional connection arguments to pass to alternate kombu channel implementations. Consult the transport documentation for available options.
- **heartbeat** – Heartbeat interval in int/float seconds. Note that if heartbeats are enabled then the `heartbeat_check()` method must be called regularly, around once per second.

---

**Note:** The connection is established lazily when needed. If you need the connection to be established, then force it by calling `connect()`:

```
>>> conn = Connection('amqp://')
>>> conn.connect()
```

and always remember to close the connection:

```
>>> conn.release()
```

---

#### Legacy options

These options have been replaced by the URL argument, but are still supported for backwards compatibility:

#### Parameters

- **hostname** – Host name/address. NOTE: You cannot specify both the URL argument and use the hostname keyword argument at the same time.
- **userid** – Default user name if not provided in the URL.
- **password** – Default password if not provided in the URL.
- **virtual\_host** – Default virtual host if not provided in the URL.



- **port** – Default port if not provided in the URL.

---

#### Attributes

**hostname** = None

**port** = None

**userid** = None

**password** = None

**virtual\_host** = '/'

**ssl** = None

**login\_method** = None

**failover\_strategy** = 'round-robin'

Strategy used to select new hosts when reconnecting after connection failure. One of "round-robin", "shuffle" or any custom iterator constantly yielding new URLs to try.

**connect\_timeout** = 5

**heartbeat** = None

Heartbeat value, currently only supported by the py-amqp transport.

**default\_channel**

Default channel, created upon access and closed when the connection is closed.

Can be used for automatic channel handling when you only need one channel, and also it is the channel implicitly used if a connection is passed instead of a channel, to functions that require a channel.

**connected**

Return true if the connection has been established.

**recoverable\_connection\_errors**

List of connection related exceptions that can be recovered from, but where the connection must be closed and re-established first.

**recoverable\_channel\_errors**

List of channel related exceptions that can be automatically recovered from without re-establishing the connection.

**connection\_errors**

List of exceptions that may be raised by the connection.

**channel\_errors**

List of exceptions that may be raised by the channel.

**transport**

**connection**

The underlying connection object.

**Warning:** This instance is transport specific, so do not depend on the interface of this object.

**uri\_prefix** = None

**declared\_entities** = None

The cache of declared entities is per connection, in case the server loses data.

**cycle = None**

Iterator returning the next broker URL to try in the event of connection failure (initialized by `failover_strategy`).

**host**

The host as a host name/port pair separated by colon.

**manager**

Experimental manager that can be used to manage/monitor the broker instance. Not available for all transports.

**supports\_heartbeats****is\_evented**

---

**Methods**

**as\_uri** (*include\_password=False, mask='\*\*', getfields=<operator.itemgetter object>*)

Convert connection parameters to URL form.

**connect** ()

Establish connection to server immediately.

**channel** ()

Create and return a new channel.

**drain\_events** (*\*\*kwargs*)

Wait for a single event from the server.

**Parameters** `timeout` – Timeout in seconds before we give up.

:raises `socket.timeout`: if the timeout is exceeded.

**release** ()

Close the connection (if open).

**autoretry** (*fun, channel=None, \*\*ensure\_options*)

Decorator for functions supporting a `channel` keyword argument.

The resulting callable will retry calling the function if it raises connection or channel related errors. The return value will be a tuple of (`retval`, `last_created_channel`).

If a `channel` is not provided, then one will be automatically acquired (remember to close it afterwards).

See `ensure()` for the full list of supported keyword arguments.

Example usage:

```
channel = connection.channel()
try:
    ret, channel = connection.autoretry(publish_messages, channel)
finally:
    channel.close()
```

**ensure\_connection** (*errback=None, max\_retries=None, interval\_start=2, interval\_step=2, interval\_max=30, callack=None*)

Ensure we have a connection to the server.

If not retry establishing the connection with the settings specified.

**Parameters**

- **errback** – Optional callback called each time the connection can't be established. Arguments provided are the exception raised and the interval that will be slept (`exc`, `interval`).
- **max\_retries** – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval\_start** – The number of seconds we start sleeping for.
- **interval\_step** – How many seconds added to the interval for each retry.
- **interval\_max** – Maximum number of seconds to sleep between each retry.
- **callback** – Optional callback that is called for every internal iteration (1 s)

**ensure** (*obj*, *fun*, *errback=None*, *max\_retries=None*, *interval\_start=1*, *interval\_step=1*, *interval\_max=1*, *on\_revive=None*)

Ensure operation completes, regardless of any channel/connection errors occurring.

Will retry by establishing the connection, and reapplying the function.

#### Parameters

- **fun** – Method to apply.
- **errback** – Optional callback called each time the connection can't be established. Arguments provided are the exception raised and the interval that will be slept (`exc`, `interval`).
- **max\_retries** – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval\_start** – The number of seconds we start sleeping for.
- **interval\_step** – How many seconds added to the interval for each retry.
- **interval\_max** – Maximum number of seconds to sleep between each retry.

#### Example

This is an example ensuring a publish operation:

```
>>> from kombu import Connection, Producer
>>> conn = Connection('amqp://')
>>> producer = Producer(conn)

>>> def errback(exc, interval):
...     logger.error('Error: %r', exc, exc_info=1)
...     logger.info('Retry in %s seconds.', interval)

>>> publish = conn.ensure(producer, producer.publish,
...                       errback=errback, max_retries=3)
>>> publish({'hello': 'world'}, routing_key='dest')
```

**revive** (*new\_channel*)

Revive connection after connection re-established.

**create\_transport** ()

**get\_transport\_cls** ()

Get the currently used transport class.

**clone** (\*\*kwargs)

Create a copy of the connection with the same connection settings.

**info ()**

Get connection info.

**switch (url)**

Switch connection parameters to use a new URL (does not reconnect)

**maybe\_switch\_next ()**

Switch to next URL given by the current failover strategy (if any).

**heartbeat\_check (rate=2)**

Allow the transport to perform any periodic tasks required to make heartbeats work. This should be called approximately every second.

If the current transport does not support heartbeats then this is a noop operation.

**Parameters rate** – Rate is how often the tick is called compared to the actual heartbeat value.

E.g. if the heartbeat is set to 3 seconds, and the tick is called every 3 / 2 seconds, then the rate is 2. This value is currently unused by any transports.

**maybe\_close\_channel (channel)**

Close given channel, but ignore connection and channel errors.

**register\_with\_event\_loop (loop)**

**close ()**

Close the connection (if open).

**\_close ()**

Really close connection, even if part of a connection pool.

**completes\_cycle (retries)**

Return true if the cycle is complete after number of *retries*.

**get\_manager (\*args, \*\*kwargs)**

**Producer (channel=None, \*args, \*\*kwargs)**

Create new *kombu.Producer* instance using this connection.

**Consumer (queues=None, channel=None, \*args, \*\*kwargs)**

Create new *kombu.Consumer* instance using this connection.

**Pool (limit=None, preload=None)**

Pool of connections.

See *ConnectionPool*.

#### Parameters

- **limit** – Maximum number of active connections. Default is no limit.
- **preload** – Number of connections to preload when the pool is created. Default is 0.

*Example usage:*

```
>>> connection = Connection('amqp://')
>>> pool = connection.Pool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "kombu/connection.py", line 354, in acquire
```

```
raise ConnectionLimitExceeded(self.limit)
kombu.exceptions.ConnectionLimitExceeded: 2
```

**ChannelPool** (*limit=None, preload=None*)

Pool of channels.

See *ChannelPool*.

#### Parameters

- **limit** – Maximum number of active channels. Default is no limit.
- **preload** – Number of channels to preload when the pool is created. Default is 0.

*Example usage:*

```
>>> connection = Connection('amqp://')
>>> pool = connection.ChannelPool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "kombu/connection.py", line 354, in acquire
    raise ChannelLimitExceeded(self.limit)
kombu.connection.ChannelLimitExceeded: 2
```

**SimpleQueue** (*name, no\_ack=None, queue\_opts=None, exchange\_opts=None, channel=None, \*\*kwargs*)

Create new *SimpleQueue*, using a channel from this connection.

If *name* is a string, a queue and exchange will be automatically created using that name as the name of the queue and exchange, also it will be used as the default routing key.

#### Parameters

- **name** – Name of the queue/or a *Queue*.
- **no\_ack** – Disable acknowledgements. Default is false.
- **queue\_opts** – Additional keyword arguments passed to the constructor of the automatically created *Queue*.
- **exchange\_opts** – Additional keyword arguments passed to the constructor of the automatically created *Exchange*.
- **channel** – Custom channel to use. If not specified the connection default channel is used.

**SimpleBuffer** (*name, no\_ack=None, queue\_opts=None, exchange\_opts=None, channel=None, \*\*kwargs*)

Create new *SimpleQueue* using a channel from this connection.

Same as *SimpleQueue()*, but configured with buffering semantics. The resulting queue and exchange will not be durable, also auto delete is enabled. Messages will be transient (not persistent), and acknowledgements are disabled (*no\_ack*).

## 4.2 Exchange

Example creating an exchange declaration:

```
>>> news_exchange = Exchange('news', type='topic')
```

For now `news_exchange` is just a declaration, you can't perform actions on it. It just describes the name and options for the exchange.

The exchange can be bound or unbound. Bound means the exchange is associated with a channel and operations can be performed on it. To bind the exchange you call the exchange with the channel as argument:

```
>>> bound_exchange = news_exchange(channel)
```

Now you can perform operations like `declare()` or `delete()`:

```
>>> bound_exchange.declare()
>>> message = bound_exchange.Message('Cure for cancer found!')
>>> bound_exchange.publish(message, routing_key='news.science')
>>> bound_exchange.delete()
```

**class** kombu.**Exchange** (*name='', type='', channel=None, \*\*kwargs*)  
An Exchange declaration.

### Parameters

- **name** – See *name*.
- **type** – See *type*.
- **channel** – See *channel*.
- **durable** – See *durable*.
- **auto\_delete** – See *auto\_delete*.
- **delivery\_mode** – See *delivery\_mode*.
- **arguments** – See *arguments*.

### name

Name of the exchange. Default is no name (the default exchange).

### type

*This description of AMQP exchange types was shamelessly stolen from the blog post ‘AMQP in 10 minutes: Part 4’ by Rajith Attapattu. Reading this article is recommended if you’re new to amqp.*

“AMQP defines four default exchange types (routing algorithms) that covers most of the common messaging use cases. An AMQP broker can also define additional exchange types, so see your broker manual for more information about available exchange types.

#### •*direct (default)*

Direct match between the routing key in the message, and the routing criteria used when a queue is bound to this exchange.

#### •*topic*

Wildcard match between the routing key and the routing pattern specified in the exchange/queue binding. The routing key is treated as zero or more words delimited by “.” and supports special wildcard characters. “\*” matches a single word and “#” matches zero or more words.

#### •*fanout*

Queues are bound to this exchange with no arguments. Hence any message sent to this exchange will be forwarded to all queues bound to this exchange.

- *headers*

Queues are bound to this exchange with a table of arguments containing headers and values (optional). A special argument named “x-match” determines the matching algorithm, where “all” implies an *AND* (all pairs must match) and “any” implies *OR* (at least one pair must match).

*arguments* is used to specify the arguments.

**channel**

The channel the exchange is bound to (if bound).

**durable**

Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged when a server restarts. Default is `True`.

**auto\_delete**

If set, the exchange is deleted when all queues have finished using it. Default is `False`.

**delivery\_mode**

The default delivery mode used for messages. The value is an integer, or alias string.

- 1 or “*transient*”

The message is transient. Which means it is stored in memory only, and is lost if the server dies or restarts.

- 2 or “**persistent**” (*default*) The message is persistent. Which means the message is stored both in-memory, and on disk, and therefore preserved if the server dies or restarts.

The default value is 2 (persistent).

**arguments**

Additional arguments to specify when the exchange is declared.

**maybe\_bind** (*channel*)

Bind instance to channel if not already bound.

**Message** (*body*, *delivery\_mode=None*, *priority=None*, *content\_type=None*, *content\_encoding=None*, *properties=None*, *headers=None*)

Create message instance to be sent with `publish()`.

**Parameters**

- **body** – Message body.
- **delivery\_mode** – Set custom delivery mode. Defaults to `delivery_mode`.
- **priority** – Message priority, 0 to 9. (currently not supported by RabbitMQ).
- **content\_type** – The messages `content_type`. If `content_type` is set, no serialization occurs as it is assumed this is either a binary object, or you’ve done your own serialization. Leave blank if using built-in serialization as our library properly sets `content_type`.
- **content\_encoding** – The character set in which this object is encoded. Use “binary” if sending in raw binary objects. Leave blank if using built-in serialization as our library properly sets `content_encoding`.
- **properties** – Message properties.
- **headers** – Message headers.

`PERSISTENT_DELIVERY_MODE = 2`

**TRANSIENT\_DELIVERY\_MODE = 1**

**attrs** = (('name', None), ('type', None), ('arguments', None), ('durable', <type 'bool'>), ('passive', <type 'bool'>), ('auto

**auto\_delete** = False

**bind\_to** (*exchange=''*, *routing\_key=''*, *arguments=None*, *nowait=False*, *\*\*kwargs*)

Binds the exchange to another exchange.

**Parameters nowait** – If set the server will not respond, and the call will not block waiting for a response. Default is `False`.

**binding** (*routing\_key=''*, *arguments=None*, *unbind\_arguments=None*)

**can\_cache\_declaration**

**declare** (*nowait=False*, *passive=None*)

Declare the exchange.

Creates the exchange on the broker.

**Parameters nowait** – If set the server will not respond, and a response will not be waited for. Default is `False`.

**delete** (*if\_unused=False*, *nowait=False*)

Delete the exchange declaration on server.

**Parameters**

- **if\_unused** – Delete only if the exchange has no bindings. Default is `False`.
- **nowait** – If set the server will not respond, and a response will not be waited for. Default is `False`.

**delivery\_mode = 2**

**durable = True**

**name = ''**

**passive = False**

**publish** (*message*, *routing\_key=None*, *mandatory=False*, *immediate=False*, *exchange=None*)

Publish message.

**Parameters**

- **message** – *Message ()* instance to publish.
- **routing\_key** – Routing key.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.

**type = 'direct'**

**unbind\_from** (*source=''*, *routing\_key=''*, *nowait=False*, *arguments=None*)

Delete previously created exchange binding from the server.

## 4.3 Queue

Example creating a queue using our exchange in the [Exchange](#) example:



```
>>> science_news = Queue('science_news',
...                       exchange=news_exchange,
...                       routing_key='news.science')
```

For now *science\_news* is just a declaration, you can't perform actions on it. It just describes the name and options for the queue.

The queue can be bound or unbound. Bound means the queue is associated with a channel and operations can be performed on it. To bind the queue you call the queue instance with the channel as an argument:

```
>>> bound_science_news = science_news(channel)
```

Now you can perform operations like `declare()` or `purge()`:

```
>>> bound_science_news.declare()
>>> bound_science_news.purge()
>>> bound_science_news.delete()
```

```
class kombu.Queue(name='', exchange=None, routing_key='', channel=None, bindings=None,
                  on_declared=None, **kwargs)
```

A Queue declaration.

#### Parameters

- **name** – See *name*.
- **exchange** – See *exchange*.
- **routing\_key** – See *routing\_key*.
- **channel** – See *channel*.
- **durable** – See *durable*.
- **exclusive** – See *exclusive*.
- **auto\_delete** – See *auto\_delete*.
- **queue\_arguments** – See *queue\_arguments*.
- **binding\_arguments** – See *binding\_arguments*.
- **on\_declared** – See *on\_declared*

#### name

Name of the queue. Default is no name (default queue destination).

#### exchange

The *Exchange* the queue binds to.

#### routing\_key

The routing key (if any), also called *binding key*.

The interpretation of the routing key depends on the *Exchange.type*.

##### •direct exchange

Matches if the routing key property of the message and the *routing\_key* attribute are identical.

##### •fanout exchange

Always matches, even if the binding does not have a key.

##### •topic exchange

Matches the routing key property of the message by a primitive pattern matching scheme. The message routing key then consists of words separated by dots (".", like domain names), and two special characters are available; star ("\*") and hash ("#"). The star matches any word, and the hash matches zero or more words. For example `*.stock.#` matches the routing keys `"usd.stock"` and `"eur.stock.db"` but not `"stock.nasdaq"`.

**channel**

The channel the Queue is bound to (if bound).

**durable**

Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue.

Default is `True`.

**exclusive**

Exclusive queues may only be consumed from by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'.

Default is `False`.

**auto\_delete**

If set, the queue is deleted when all consumers have finished using it. Last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted.

**queue\_arguments**

Additional arguments used when declaring the queue.

**binding\_arguments**

Additional arguments used when binding the queue.

**alias**

Unused in Kombu, but applications can take advantage of this. For example to give alternate names to queues with automatically generated queue names.

**on\_declared**

Optional callback to be applied when the queue has been declared (the `queue_declare` operation is complete). This must be a function with a signature that accepts at least 3 positional arguments: (`name`, `messages`, `consumers`).

**maybe\_bind** (*channel*)

Bind instance to channel if not already bound.

**exception ContentDisallowed**

Consumer does not allow this content-type.

`Queue.as_dict` (*recurse=False*)

`Queue.attrs` = (('name', None), ('exchange', None), ('routing\_key', None), ('queue\_arguments', None), ('binding\_argu

`Queue.auto_delete` = `False`

`Queue.bind` (*channel*)

`Queue.bind_to` (*exchange='', routing\_key='', arguments=None, nowait=False*)

`Queue.can_cache_declaration`

`Queue.cancel` (*consumer\_tag*)

Cancel a consumer by consumer tag.

`Queue.consume` (*consumer\_tag=''*, *callback=None*, *no\_ack=None*, *nowait=False*)  
Start a queue consumer.

Consumers last as long as the channel they were created on, or until the client cancels them.

#### Parameters

- **consumer\_tag** – Unique identifier for the consumer. The consumer tag is local to a connection, so two clients can use the same consumer tags. If this field is empty the server will generate a unique tag.
- **no\_ack** – If enabled the broker will automatically ack messages.
- **nowait** – Do not wait for a reply.
- **callback** – callback called for each delivered message

`Queue.declare` (*nowait=False*)  
Declares the queue, the exchange and binds the queue to the exchange.

`Queue.delete` (*if\_unused=False*, *if\_empty=False*, *nowait=False*)  
Delete the queue.

#### Parameters

- **if\_unused** – If set, the server will only delete the queue if it has no consumers. A channel error will be raised if the queue has consumers.
- **if\_empty** – If set, the server will only delete the queue if it is empty. If it is not empty a channel error will be raised.
- **nowait** – Do not wait for a reply.

`Queue.durable = True`

`Queue.exchange = <unbound Exchange '(direct)>`

`Queue.exclusive = False`

`classmethod Queue.from_dict` (*queue*, *\*\*options*)

`Queue.get` (*no\_ack=None*, *accept=None*)  
Poll the server for a new message.

Must return the message if a message was available, or `None` otherwise.

#### Parameters

- **no\_ack** – If enabled the broker will automatically ack messages.
- **accept** – Custom list of accepted content types.

This method provides direct access to the messages in a queue using a synchronous dialogue, designed for specific types of applications where synchronous functionality is more important than performance.

`Queue.name = ''`

`Queue.no_ack = False`

`Queue.purge` (*nowait=False*)  
Remove all ready messages from the queue.

`Queue.queue_bind` (*nowait=False*)  
Create the queue binding on the server.

`Queue.queue_declare` (*nowait=False*, *passive=False*)  
Declare queue on the server.

### Parameters

- **nowait** – Do not wait for a reply.
- **passive** – If set, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state.

`Queue.queue_unbind(arguments=None, nowait=False)`

`Queue.routing_key = ''`

`Queue.unbind_from(exchange='', routing_key='', arguments=None, nowait=False)`  
Unbind queue by deleting the binding from the server.

`Queue.when_bound()`

## 4.4 Message Producer

```
class kombu.Producer(channel, exchange=None, routing_key=None, serializer=None,
                    auto_declare=None, compression=None, on_return=None)
    Message Producer.
```

### Parameters

- **channel** – Connection or channel.
- **exchange** – Optional default exchange.
- **routing\_key** – Optional default routing key.
- **serializer** – Default serializer. Default is “*json*”.
- **compression** – Default compression method. Default is no compression.
- **auto\_declare** – Automatically declare the default exchange at instantiation. Default is `True`.
- **on\_return** – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to `publish()` is used. This callback needs the following signature: (*exception*, *exchange*, *routing\_key*, *message*). Note that the producer needs to drain events to use this feature.

**channel**

**exchange = None**  
Default exchange

**routing\_key = ''**  
Default routing key.

**serializer = None**  
Default serializer to use. Default is JSON.

**compression = None**  
Default compression method. Disabled by default.

**auto\_declare = True**  
By default the exchange is declared at instantiation. If you want to declare manually then you can set this to `False`.

**on\_return = None**  
Basic return callback.

**connection**

**declare** ()

Declare the exchange.

This happens automatically at instantiation if *auto\_declare* is enabled.

**maybe\_declare** (*entity*, *retry=False*, *\*\*retry\_policy*)

Declare the exchange if it hasn't already been declared during this session.

**publish** (*body*, *routing\_key=None*, *delivery\_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content\_type=None*, *content\_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry\_policy=None*, *declare=[]*, *expiration=None*, *\*\*properties*)

Publish message to the specified exchange.

#### Parameters

- **body** – Message body.
- **routing\_key** – Message routing key.
- **delivery\_mode** – See *delivery\_mode*.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content\_type** – Content type. Default is auto-detect.
- **content\_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using *maybe\_declare()*.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry\_policy** – Retry configuration, this is the keywords supported by *ensure()*.
- **expiration** – A TTL in seconds can be specified per message. Default is no expiration.
- **\*\*properties** – Additional message properties, see AMQP spec.

**revive** (*channel*)

Revive the producer after connection loss.

## 4.5 Message Consumer

**class** kombu.Consumer (*channel*, *queues=None*, *no\_ack=None*, *auto\_declare=None*, *callbacks=None*, *on\_decode\_error=None*, *on\_message=None*, *accept=None*, *tag\_prefix=None*)

Message consumer.

#### Parameters

- **channel** – see *channel*.

- **queues** – see *queues*.
- **no\_ack** – see *no\_ack*.
- **auto\_declare** – see *auto\_declare*
- **callbacks** – see *callbacks*.
- **on\_message** – See *on\_message*
- **on\_decode\_error** – see *on\_decode\_error*.

**channel = None**

The connection/channel to use for this consumer.

**queues = None**

A single *Queue*, or a list of queues to consume from.

**no\_ack = None**

Flag for automatic message acknowledgment. If enabled the messages are automatically acknowledged by the broker. This can increase performance but means that you have no control of when the message is removed.

Disabled by default.

**auto\_declare = True**

By default all entities will be declared at instantiation, if you want to handle this manually you can set this to `False`.

**callbacks = None**

List of callbacks called in order when a message is received.

The signature of the callbacks must take two arguments: (*body*, *message*), which is the decoded message body and the *Message* instance (a subclass of *Message*).

**on\_message = None**

Optional function called whenever a message is received.

When defined this function will be called instead of the *receive()* method, and *callbacks* will be disabled.

So this can be used as an alternative to *callbacks* when you don't want the body to be automatically decoded. Note that the message will still be decompressed if the message has the `compression` header set.

The signature of the callback must take a single argument, which is the raw message object (a subclass of *Message*).

Also note that the `message.body` attribute, which is the raw contents of the message body, may in some cases be a read-only `buffer` object.

**on\_decode\_error = None**

Callback called when a message can't be decoded.

The signature of the callback must take two arguments: (*message*, *exc*), which is the message that can't be decoded and the exception that occurred while trying to decode it.

**connection**

**declare ()**

Declare queues, exchanges and bindings.

This is done automatically at instantiation if *auto\_declare* is set.

**register\_callback** (*callback*)

Register a new callback to be called when a message is received.

The signature of the callback needs to accept two arguments: (*body*, *message*), which is the decoded message body and the *Message* instance (a subclass of *Message*).

**add\_queue** (*queue*)

Add a queue to the list of queues to consume from.

This will not start consuming from the queue, for that you will have to call *consume()* after.

**add\_queue\_from\_dict** (*queue*, *\*\*options*)

This method is deprecated.

Instead please use:

```
consumer.add_queue(Queue.from_dict(d))
```

**consume** (*no\_ack=None*)

Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues ( use *cancel\_by\_queue()*).

**Parameters** *no\_ack* – See *no\_ack*.

**cancel** ()

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

**cancel\_by\_queue** (*queue*)

Cancel consumer by queue name.

**consuming\_from** (*queue*)

Return *True* if the consumer is currently consuming from *queue*'.

**purge** ()

Purge messages from all queues.

**Warning:** This will *delete all ready messages*, there is no undo operation.

**flow** (*active*)

Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

**qos** (*prefetch\_size=0*, *prefetch\_count=0*, *apply\_global=False*)

Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the *no\_ack* option is set.

**Parameters**

- **prefetch\_size** – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning “no specific limit”, although other prefetch limits may still apply.
- **prefetch\_count** – Specify the prefetch window in terms of whole messages.
- **apply\_global** – Apply new settings globally on all channels.

**recover** (*requeue=False*)

Redeliver unacknowledged messages.

Asks the broker to redeliver all unacknowledged messages on the specified channel.

**Parameters requeue** – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

**receive** (*body, message*)

Method called when a message is received.

This dispatches to the registered *callbacks*.

**Parameters**

- **body** – The decoded message body.
- **message** – The *Message* instance.

**Raises NotImplementedError** – If no consumer callbacks have been registered.

**revive** (*channel*)

Revive consumer after connection loss.

## 4.6 Common Utilities - kombu.common

- *kombu.common*

### 4.6.1 kombu.common

Common Utilities.

```
class kombu.common.Broadcast (name=None, queue=None, auto_delete=True, exchange=None, alias=None, **kwargs)
```

Convenience class used to define broadcast queues.

Every queue instance will have a unique name, and both the queue and exchange is configured with auto deletion.

**Parameters**

- **name** – This is used as the name of the exchange.
- **queue** – By default a unique id is used for the queue name for every consumer. You can specify a custom queue name here.
- **\*\*kwargs** – See *Queue* for a list of additional keyword arguments supported.

```
attrs = (('name', None), ('exchange', None), ('routing_key', None), ('queue_arguments', None), ('binding_arguments', None))
```

```
kombu.common.maybe_declare (entity, channel=None, retry=False, **retry_policy)
```



`kombu.common.uuid()`

Generate a unique id, having - hopefully - a very small chance of collision.

For now this is provided by `uuid.uuid4()`.

`kombu.common.ittermessages` (*conn, channel, queue, limit=1, timeout=None, callbacks=None, \*\*kwargs*)

`kombu.common.send_reply` (*exchange, req, msg, producer=None, retry=False, retry\_policy=None, \*\*props*)

Send reply for request.

#### Parameters

- **exchange** – Reply exchange
- **req** – Original request, a message with a `reply_to` property.
- **producer** – Producer instance
- **retry** – If true must retry according to `reply_policy` argument.
- **retry\_policy** – Retry settings.
- **props** – Extra properties

`kombu.common.collect_replies` (*conn, channel, queue, \*args, \*\*kwargs*)

Generator collecting replies from queue

`kombu.common.insured` (*pool, fun, args, kwargs, errback=None, on\_revive=None, \*\*opts*)

Ensures function performing broker commands completes despite intermittent connection failures.

`kombu.common.drain_consumer` (*consumer, limit=1, timeout=None, callbacks=None*)

`kombu.common.eventloop` (*conn, limit=None, timeout=None, ignore\_timeouts=False*)

Best practice generator wrapper around `Connection.drain_events`.

Able to drain events forever, with a limit, and optionally ignoring timeout errors (a timeout of 1 is often used in environments where the socket can get “stuck”, and is a best practice for Kombu consumers).

#### Examples

`eventloop` is a generator:

```
from kombu.common import eventloop

def run(connection):
    it = eventloop(connection, timeout=1, ignore_timeouts=True)
    next(it) # one event consumed, or timed out.

    for _ in eventloop(connection, timeout=1, ignore_timeouts=True):
        pass # loop forever.
```

It also takes an optional limit parameter, and timeout errors are propagated by default:

```
for _ in eventloop(connection, limit=1, timeout=1):
    pass
```

See also:

`ittermessages()`, which is an event loop bound to one or more consumers, that yields any messages received.

## 4.7 Mixin Classes - kombu.mixins

- *kombu.mixins*

### 4.7.1 kombu.mixins

Useful mixin classes.

**class** `kombu.mixins.ConsumerMixin`

Convenience mixin for implementing consumer programs.

It can be used outside of threads, with threads, or greenthreads (eventlet/gevent) too.

The basic class would need a `connection` attribute which must be a `Connection` instance, and define a `get_consumers()` method that returns a list of `kombu.Consumer` instances to use. Supporting multiple consumers is important so that multiple channels can be used for different QoS requirements.

**Example:**

```
class Worker(ConsumerMixin):
    task_queue = Queue('tasks', Exchange('tasks'), 'tasks')

    def __init__(self, connection):
        self.connection = None

    def get_consumers(self, Consumer, channel):
        return [Consumer(queues=[self.task_queue],
                        callbacks=[self.on_task])]

    def on_task(self, body, message):
        print('Got task: {0!r}'.format(body))
        message.ack()
```

**Additional handler methods:**

- `extra_context()`

Optional extra context manager that will be entered after the connection and consumers have been set up.

Takes arguments (`connection`, `channel`).

- `on_connection_error()`

Handler called if the connection is lost/ or is unavailable.

Takes arguments (`exc`, `interval`), where `interval` is the time in seconds when the connection will be retried.

The default handler will log the exception.

- `on_connection_revived()`

Handler called as soon as the connection is re-established after connection failure.

Takes no arguments.

- `on_consume_ready()`

Handler called when the consumer is ready to accept messages.

Takes arguments (*connection*, *channel*, *consumers*). Also keyword arguments to consume are forwarded to this handler.

- `on_consume_end()`

Handler called after the consumers are cancelled. Takes arguments (*connection*, *channel*).

- `on_iteration()`

Handler called for every iteration while draining events.

Takes no arguments.

- `on_decode_error()`

Handler called if a consumer was unable to decode the body of a message.

Takes arguments (*message*, *exc*) where *message* is the original message object.

The default handler will log the error and acknowledge the message, so if you override make sure to call `super`, or perform these steps yourself.

**Consumer** (*\*args*, *\*\*kws*)

**channel\_errors**

**connect\_max\_retries** = None

maximum number of retries trying to re-establish the connection, if the connection is lost/unavailable.

**connection\_errors**

**consume** (*limit=None*, *timeout=None*, *safety\_interval=1*, *\*\*kwargs*)

**consumer\_context** (*\*args*, *\*\*kws*)

**create\_connection** ()

**establish\_connection** (*\*args*, *\*\*kws*)

**extra\_context** (*\*args*, *\*\*kws*)

**get\_consumers** (*Consumer*, *channel*)

**maybe\_conn\_error** (*fun*)

Use `kombu.common.ignore_errors()` instead.

**on\_connection\_error** (*exc*, *interval*)

**on\_connection\_revived** ()

**on\_consume\_end** (*connection*, *channel*)

**on\_consume\_ready** (*connection*, *channel*, *consumers*, *\*\*kwargs*)

**on\_decode\_error** (*message*, *exc*)

**on\_iteration** ()

**restart\_limit**

**run** (*\_tokens=1*)

**should\_stop** = False

When this is set to true the consumer should stop consuming and return, so that it can be joined if it is the implementation of a thread.

## 4.8 kombu.simple

Simple interface.

- *Persistent*
- *Buffer*

### 4.8.1 Persistent

```
class kombu.simple.SimpleQueue(channel, name, no_ack=None, queue_opts=None, exchange_opts=None, serializer=None, compression=None, **kwargs)
```

**channel**

Current channel

**producer**

*Producer* used to publish messages.

**consumer**

*Consumer* used to receive messages.

**no\_ack**

flag to enable/disable acknowledgements.

**queue**

*Queue* to consume from (if consuming).

**queue\_opts**

Additional options for the queue declaration.

**exchange\_opts**

Additional options for the exchange declaration.

**get** (*block=True, timeout=None*)

**get\_nowait** ()

**put** (*message, serializer=None, headers=None, compression=None, routing\_key=None, \*\*kwargs*)

**clear** ()

**\_\_len\_\_** ()

*len(self) -> self.qsize()*

**qsize** ()

**close** ()

### 4.8.2 Buffer

```
class kombu.simple.SimpleBuffer(channel, name, no_ack=None, queue_opts=None, exchange_opts=None, serializer=None, compression=None, **kwargs)
```

**channel**

Current channel

**producer**

*Producer* used to publish messages.

**consumer**

*Consumer* used to receive messages.

**no\_ack**

flag to enable/disable acknowledgements.

**queue**

*Queue* to consume from (if consuming).

**queue\_opts**

Additional options for the queue declaration.

**exchange\_opts**

Additional options for the exchange declaration.

**get** (*block=True, timeout=None*)

**get\_nowait** ()

**put** (*message, serializer=None, headers=None, compression=None, routing\_key=None, \*\*kwargs*)

**clear** ()

**\_\_len\_\_** ()

*len(self) -> self.qsize()*

**qsize** ()

**close** ()

## 4.9 Clocks and Synchronization - kombu.clocks

- *kombu.clocks*

### 4.9.1 kombu.clocks

Logical Clocks and Synchronization.

**class** `kombu.clocks.LamportClock` (*initial\_value=0, Lock=<built-in function allocate\_lock>*)  
Lamport's logical clock.

From Wikipedia:

A Lamport logical clock is a monotonically incrementing software counter maintained in each process. It follows some simple rules:

- A process increments its counter before each event in that process;
- When a process sends a message, it includes its counter value with the message;
- On receiving a message, the receiver process sets its counter to be greater than the maximum of its own value and the received value before it considers the message received.

Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with the sender.

**See also:**

- [Lamport timestamps](#)
- [Lamports distributed mutex](#)

*Usage*

When sending a message use `forward()` to increment the clock, when receiving a message use `adjust()` to sync with the time stamp of the incoming message.

**adjust** (*other*)

**forward** ()

**sort\_heap** (*h*)

List of tuples containing at least two elements, representing an event, where the first element is the event's scalar clock value, and the second element is the id of the process (usually "hostname:pid"): `sh([(clock, processid, ...?), (...)])`

The list must already be sorted, which is why we refer to it as a heap.

The tuple will not be unpacked, so more than two elements can be present.

Will return the latest event.

**value = 0**

The clocks current value.

**class** kombu.clocks.timetuple

Tuple of event clock information.

Can be used as part of a heap to keep events ordered.

**Parameters**

- **clock** – Event clock value.
- **timestamp** – Event UNIX timestamp value.
- **id** – Event host id (e.g. hostname:pid).
- **obj** – Optional obj to associate with this event.

**clock**

itemgetter(item, ...) -> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`. After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2], r[5], r[3])`

**id**

itemgetter(item, ...) -> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`. After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2], r[5], r[3])`

**obj**

itemgetter(item, ...) -> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`. After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2], r[5], r[3])`

**timestamp**

itemgetter(item, ...) -> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`. After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2], r[5], r[3])`

## 4.10 kombu.compat

Carrot compatible interface for *Publisher* and *Producer*.

See <http://packages.python.org/pyqi/carrot> for documentation.

- *Publisher*
- *Consumer*
- *ConsumerSet*

### 4.10.1 Publisher

Replace with `kombu.Producer`.

```
class kombu.compat.Publisher(connection, exchange=None, routing_key=None,
                             change_type=None, durable=None, auto_delete=None,
                             channel=None, **kwargs)
```

**auto\_declare = True**

**auto\_delete = False**

**backend**

**channel**

**close()**

**compression = None**

**connection**

**declare()**

Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

**durable = True**

**exchange = ''**

**exchange\_type = 'direct'**

**maybe\_declare** (*entity*, *retry=False*, *\*\*retry\_policy*)

Declare the exchange if it hasn't already been declared during this session.

**on\_return = None**

**publish** (*body*, *routing\_key=None*, *delivery\_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content\_type=None*, *content\_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry\_policy=None*, *declare=[]*, *expiration=None*, *\*\*properties*)

Publish message to the specified exchange.

### Parameters

- **body** – Message body.
- **routing\_key** – Message routing key.
- **delivery\_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content\_type** – Content type. Default is auto-detect.
- **content\_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry\_policy** – Retry configuration, this is the keywords supported by `ensure()`.
- **expiration** – A TTL in seconds can be specified per message. Default is no expiration.
- **\*\*properties** – Additional message properties, see AMQP spec.

**release()**

**revive(channel)**

Revive the producer after connection loss.

**routing\_key = ''**

**send(\*args, \*\*kwargs)**

**serializer = None**

## 4.10.2 Consumer

Replace with `kombu.Consumer`.

```
class kombu.compat.Consumer(connection, queue=None, exchange=None, routing_key=None,
                             exchange_type=None, durable=None, exclusive=None,
                             auto_delete=None, **kwargs)
```

**exception ContentDisallowed**

Consumer does not allow this content-type.

**args**

**message**

`Consumer.accept = None`



`Consumer.add_queue(queue)`

Add a queue to the list of queues to consume from.

This will not start consuming from the queue, for that you will have to call `consume()` after.

`Consumer.add_queue_from_dict(queue, **options)`

This method is deprecated.

Instead please use:

```
consumer.add_queue(Queue.from_dict(d))
```

`Consumer.auto_declare = True`

`Consumer.auto_delete = False`

`Consumer.callbacks = None`

`Consumer.cancel()`

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

`Consumer.cancel_by_queue(queue)`

Cancel consumer by queue name.

`Consumer.channel = None`

`Consumer.close()`

`Consumer.connection`

`Consumer.consume(no_ack=None)`

Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues ( use `cancel_by_queue()` ).

**Parameters** `no_ack` – See `no_ack`.

`Consumer.consuming_from(queue)`

Return True if the consumer is currently consuming from queue'.

`Consumer.declare()`

Declare queues, exchanges and bindings.

This is done automatically at instantiation if `auto_declare` is set.

`Consumer.discard_all(filterfunc=None)`

`Consumer.durable = True`

`Consumer.exchange = ''`

`Consumer.exchange_type = 'direct'`

`Consumer.exclusive = False`

`Consumer.fetch(no_ack=None, enable_callbacks=False)`

`Consumer.flow(active)`

Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

`Consumer.iterconsume` (*limit=None, no\_ack=None*)

`Consumer.iterqueue` (*limit=None, infinite=False*)

`Consumer.no_ack = None`

`Consumer.on_decode_error = None`

`Consumer.on_message = None`

`Consumer.process_next` ()

`Consumer.purge` ()

Purge messages from all queues.

**Warning:** This will *delete all ready messages*, there is no undo operation.

`Consumer.qos` (*prefetch\_size=0, prefetch\_count=0, apply\_global=False*)

Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the *no\_ack* option is set.

#### Parameters

- **prefetch\_size** – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning “no specific limit”, although other prefetch limits may still apply.
- **prefetch\_count** – Specify the prefetch window in terms of whole messages.
- **apply\_global** – Apply new settings globally on all channels.

`Consumer.queue = ‘`

`Consumer.queues = None`

`Consumer.receive` (*body, message*)

Method called when a message is received.

This dispatches to the registered *callbacks*.

#### Parameters

- **body** – The decoded message body.
- **message** – The *Message* instance.

**Raises** `NotImplementedError` – If no consumer callbacks have been registered.

`Consumer.recover` (*requeue=False*)

Redeliver unacknowledged messages.

Asks the broker to redeliver all unacknowledged messages on the specified channel.

**Parameters** **requeue** – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

`Consumer.register_callback (callback)`

Register a new callback to be called when a message is received.

The signature of the callback needs to accept two arguments: (*body*, *message*), which is the decoded message body and the *Message* instance (a subclass of *Message*).

`Consumer.revive (channel)`

`Consumer.routing_key = ''`

`Consumer.wait (limit=None)`

### 4.10.3 ConsumerSet

Replace with `kombu.Consumer`.

```
class kombu.compat.ConsumerSet (connection, from_dict=None, consumers=None, channel=None,
                                **kwargs)
```

**exception ContentDisallowed**

Consumer does not allow this content-type.

**args**

**message**

`ConsumerSet.accept = None`

`ConsumerSet.add_consumer (consumer)`

`ConsumerSet.add_consumer_from_dict (queue, **options)`

`ConsumerSet.add_queue (queue)`

Add a queue to the list of queues to consume from.

This will not start consuming from the queue, for that you will have to call `consume ()` after.

`ConsumerSet.add_queue_from_dict (queue, **options)`

This method is deprecated.

Instead please use:

```
consumer.add_queue (Queue.from_dict (d))
```

`ConsumerSet.auto_declare = True`

`ConsumerSet.callbacks = None`

`ConsumerSet.cancel ()`

End all active queue consumers.

This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

`ConsumerSet.cancel_by_queue (queue)`

Cancel consumer by queue name.

`ConsumerSet.channel = None`

`ConsumerSet.close ()`

`ConsumerSet.connection`

`ConsumerSet.consume` (*no\_ack=None*)

Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues ( use `cancel_by_queue()`).

**Parameters** `no_ack` – See `no_ack`.

`ConsumerSet.consuming_from` (*queue*)

Return True if the consumer is currently consuming from queue'.

`ConsumerSet.declare` ()

Declare queues, exchanges and bindings.

This is done automatically at instantiation if `auto_declare` is set.

`ConsumerSet.discard_all` ()

`ConsumerSet.flow` (*active*)

Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

`ConsumerSet.iterconsume` (*limit=None, no\_ack=False*)

`ConsumerSet.no_ack = None`

`ConsumerSet.on_decode_error = None`

`ConsumerSet.on_message = None`

`ConsumerSet.purge` ()

Purge messages from all queues.

<b>Warning:</b> This will <i>delete all ready messages</i> , there is no undo operation.
--

`ConsumerSet.qos` (*prefetch\_size=0, prefetch\_count=0, apply\_global=False*)

Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the `no_ack` option is set.

**Parameters**

- **prefetch\_size** – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning “no specific limit”, although other prefetch limits may still apply.
- **prefetch\_count** – Specify the prefetch window in terms of whole messages.
- **apply\_global** – Apply new settings globally on all channels.

`ConsumerSet.queues = None`

`ConsumerSet.receive` (*body, message*)

Method called when a message is received.

This dispatches to the registered *callbacks*.

#### Parameters

- **body** – The decoded message body.
- **message** – The *Message* instance.

**Raises** `NotImplementedError` – If no consumer callbacks have been registered.

`ConsumerSet.recover` (*requeue=False*)  
Redeliver unacknowledged messages.

Asks the broker to redeliver all unacknowledged messages on the specified channel.

**Parameters** **requeue** – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

`ConsumerSet.register_callback` (*callback*)  
Register a new callback to be called when a message is received.

The signature of the callback needs to accept two arguments: (*body*, *message*), which is the decoded message body and the *Message* instance (a subclass of *Message*).

`ConsumerSet.revive` (*channel*)

## 4.11 kombu.pidbox

Generic process mailbox.

- *Introduction*
  - *Creating the applications Mailbox*
  - *Example Node*
  - *Example Client*
- *Mailbox*
- *Node*

### 4.11.1 Introduction

#### Creating the applications Mailbox

```
>>> mailbox = pidbox.Mailbox("celerybeat", type="direct")

>>> @mailbox.handler
>>> def reload_schedule(state, **kwargs):
...     state["beat"].reload_schedule()

>>> @mailbox.handler
>>> def connection_info(state, **kwargs):
...     return {"connection": state["connection"].info() }
```

### Example Node

```
>>> connection = kombu.Connection()
>>> state = {"beat": beat,
            "connection": connection}
>>> consumer = mailbox(connection).Node(hostname).listen()
>>> try:
...     while True:
...         connection.drain_events(timeout=1)
...     finally:
...         consumer.cancel()
```

### Example Client

```
>>> mailbox.cast("reload_schedule") # cast is async.
>>> info = celerybeat.call("connection_info", timeout=1)
```

## 4.11.2 Mailbox

**class** kombu.pidbox.**Mailbox** (*namespace, type='direct', connection=None, clock=None, accept=None, serializer=None*)

**namespace = None**

Name of application.

**connection = None**

Connection (if bound).

**type = 'direct'**

Exchange type (usually direct, or fanout for broadcast).

**exchange = None**

mailbox exchange (init by constructor).

**reply\_exchange = None**

exchange to send replies to.

**Node** (*hostname=None, state=None, channel=None, handlers=None*)

**call** (*destination, command, kwargs={}, timeout=None, callback=None, channel=None*)

**cast** (*destination, command, kwargs={}*)

**abcast** (*command, kwargs={}*)

**multi\_call** (*command, kwargs={}, timeout=1, limit=None, callback=None, channel=None*)

**get\_reply\_queue** ()

**get\_queue** (*hostname*)

## 4.11.3 Node

**class** kombu.pidbox.**Node** (*hostname, state=None, channel=None, handlers=None, mailbox=None*)

**hostname = None**

hostname of the node.

**mailbox = None**  
the *Mailbox* this is a node for.

**handlers = None**  
map of method name/handlers.

**state = None**  
current context (passed on to handlers)

**channel = None**  
current channel.

**Consumer** (*channel=None, no\_ack=True, accept=None, \*\*options*)

**handler** (*fun*)

**listen** (*channel=None, callback=None*)

**dispatch** (*method, arguments=None, reply\_to=None, ticket=None, \*\*kwargs*)

**dispatch\_from\_message** (*body, message=None*)

**handle\_call** (*method, arguments*)

**handle\_cast** (*method, arguments*)

**handle** (*method, arguments={}*)

**handle\_message** (*body, message=None*)

**reply** (*data, exchange, routing\_key, ticket, \*\*kwargs*)

## 4.12 kombu.exceptions

Exceptions.

**exception** `kombu.exceptions.NotBoundError`  
Trying to call channel dependent method on unbound entity.

**exception** `kombu.exceptions.MessageStateError`  
The message has already been acknowledged.

`kombu.exceptions.TimeoutError`  
alias of `timeout`

**exception** `kombu.exceptions.LimitExceeded`  
Limit exceeded.

**exception** `kombu.exceptions.ConnectionLimitExceeded`  
Maximum number of simultaneous connections exceeded.

**exception** `kombu.exceptions.ChannelsLimitExceeded`  
Maximum number of simultaneous channels exceeded.

## 4.13 Logging - kombu.log

**class** `kombu.log.LogMixin`

**annotate** (*text*)

```

critical (*args, **kwargs)
debug (*args, **kwargs)
error (*args, **kwargs)
get_logger()
get_loglevel (level)
info (*args, **kwargs)
is_enabled_for (level)
log (severity, *args, **kwargs)
logger
logger_name
warn (*args, **kwargs)

```

```
kombu.log.get_loglevel (level)
```

```
kombu.log.setup_logging (loglevel=None, logfile=None)
```

## 4.14 kombu.connection

Broker connection and pools.

- *Connection*
- *Pools*

### 4.14.1 Connection

```

class kombu.connection.Connection (hostname='localhost', userid=None, password=None, virtual_host=None, port=None, insist=False, ssl=False, transport=None, connect_timeout=5, transport_options=None, login_method=None, uri_prefix=None, heartbeat=0, failover_strategy='round-robin', alternates=None, **kwargs)

```

A connection to the broker.

**Parameters** **URL** – Broker URL, or a list of URLs, e.g.

```

Connection('amqp://guest:guest@localhost:5672//')
Connection('amqp://foo;amqp://bar', failover_strategy='round-robin')
Connection('redis://', transport_options={
    'visibility_timeout': 3000,
})

import ssl
Connection('amqp://', login_method='EXTERNAL', ssl={
    'ca_certs': '/etc/pki/tls/certs/something.crt',
    'keyfile': '/etc/something/system.key',
    'certfile': '/etc/something/system.cert',
    'cert_reqs': ssl.CERT_REQUIRED,
})

```



---

## SSL compatibility

SSL currently only works with the py-amqp, amqplib, and qpid transports. For other transports you can use stunnel.

---

### Parameters

- **ssl** – Use SSL to connect to the server. Default is `False`. May not be supported by the specified transport.
- **transport** – Default transport if not specified in the URL.
- **connect\_timeout** – Timeout in seconds for connecting to the server. May not be supported by the specified transport.
- **transport\_options** – A dict of additional connection arguments to pass to alternate kombu channel implementations. Consult the transport documentation for available options.
- **heartbeat** – Heartbeat interval in int/float seconds. Note that if heartbeats are enabled then the `heartbeat_check()` method must be called regularly, around once per second.

---

**Note:** The connection is established lazily when needed. If you need the connection to be established, then force it by calling `connect()`:

```
>>> conn = Connection('amqp://')
>>> conn.connect()
```

and always remember to close the connection:

```
>>> conn.release()
```

---

### Legacy options

These options have been replaced by the URL argument, but are still supported for backwards compatibility:

### Parameters

- **hostname** – Host name/address. NOTE: You cannot specify both the URL argument and use the hostname keyword argument at the same time.
- **userid** – Default user name if not provided in the URL.
- **password** – Default password if not provided in the URL.
- **virtual\_host** – Default virtual host if not provided in the URL.
- **port** – Default port if not provided in the URL.

### ChannelPool (*limit=None, preload=None*)

Pool of channels.

See `ChannelPool`.

### Parameters

- **limit** – Maximum number of active channels. Default is no limit.
- **preload** – Number of channels to preload when the pool is created. Default is 0.

*Example usage:*

```

>>> connection = Connection('amqp://')
>>> pool = connection.ChannelPool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "kombu/connection.py", line 354, in acquire
    raise ChannelLimitExceeded(self.limit)
kombu.connection.ChannelLimitExceeded: 2

```

**Consumer** (*queues=None, channel=None, \*args, \*\*kwargs*)  
Create new *kombu.Consumer* instance using this connection.

**Pool** (*limit=None, preload=None*)  
Pool of connections.

See *ConnectionPool*.

#### Parameters

- **limit** – Maximum number of active connections. Default is no limit.
- **preload** – Number of connections to preload when the pool is created. Default is 0.

*Example usage:*

```

>>> connection = Connection('amqp://')
>>> pool = connection.Pool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "kombu/connection.py", line 354, in acquire
    raise ConnectionLimitExceeded(self.limit)
kombu.exceptions.ConnectionLimitExceeded: 2

```

**Producer** (*channel=None, \*args, \*\*kwargs*)  
Create new *kombu.Producer* instance using this connection.

**SimpleBuffer** (*name, no\_ack=None, queue\_opts=None, exchange\_opts=None, channel=None, \*\*kwargs*)  
Create new *SimpleQueue* using a channel from this connection.

Same as *SimpleQueue()*, but configured with buffering semantics. The resulting queue and exchange will not be durable, also auto delete is enabled. Messages will be transient (not persistent), and acknowledgements are disabled (*no\_ack*).

**SimpleQueue** (*name, no\_ack=None, queue\_opts=None, exchange\_opts=None, channel=None, \*\*kwargs*)  
Create new *SimpleQueue*, using a channel from this connection.

If *name* is a string, a queue and exchange will be automatically created using that name as the name of the queue and exchange, also it will be used as the default routing key.

#### Parameters

- **name** – Name of the queue/or a *Queue*.
- **no\_ack** – Disable acknowledgements. Default is false.
- **queue\_opts** – Additional keyword arguments passed to the constructor of the automatically created *Queue*.
- **exchange\_opts** – Additional keyword arguments passed to the constructor of the automatically created *Exchange*.
- **channel** – Custom channel to use. If not specified the connection default channel is used.

**as\_uri** (*include\_password=False, mask='\*\*', getfields=<operator.itemgetter object>*)

Convert connection parameters to URL form.

**autoretry** (*fun, channel=None, \*\*ensure\_options*)

Decorator for functions supporting a `channel` keyword argument.

The resulting callable will retry calling the function if it raises connection or channel related errors. The return value will be a tuple of (`retval`, `last_created_channel`).

If a `channel` is not provided, then one will be automatically acquired (remember to close it afterwards).

See *ensure()* for the full list of supported keyword arguments.

Example usage:

```
channel = connection.channel()
try:
    ret, channel = connection.autoretry(publish_messages, channel)
finally:
    channel.close()
```

**channel()**

Create and return a new channel.

**channel\_errors**

List of exceptions that may be raised by the channel.

**clone** (*\*\*kwargs*)

Create a copy of the connection with the same connection settings.

**close()**

Close the connection (if open).

**collect** (*socket\_timeout=None*)

**completes\_cycle** (*retries*)

Return true if the cycle is complete after number of *retries*.

**connect()**

Establish connection to server immediately.

**connect\_timeout = 5**

**connected**

Return true if the connection has been established.

**connection**

The underlying connection object.

**Warning:** This instance is transport specific, so do not depend on the interface of this object.

**connection\_errors**

List of exceptions that may be raised by the connection.

**create\_transport ()****cycle = None**

Iterator returning the next broker URL to try in the event of connection failure (initialized by *failover\_strategy*).

**declared\_entities = None**

The cache of declared entities is per connection, in case the server loses data.

**default\_channel**

Default channel, created upon access and closed when the connection is closed.

Can be used for automatic channel handling when you only need one channel, and also it is the channel implicitly used if a connection is passed instead of a channel, to functions that require a channel.

**drain\_events (\*\*kwargs)**

Wait for a single event from the server.

**Parameters** **timeout** – Timeout in seconds before we give up.

:raises `socket.timeout`: if the timeout is exceeded.

**ensure (obj, fun, errback=None, max\_retries=None, interval\_start=1, interval\_step=1, interval\_max=1, on\_revive=None)**

Ensure operation completes, regardless of any channel/connection errors occurring.

Will retry by establishing the connection, and reapplying the function.

**Parameters**

- **fun** – Method to apply.
- **errback** – Optional callback called each time the connection can't be established. Arguments provided are the exception raised and the interval that will be slept (`exc, interval`).
- **max\_retries** – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval\_start** – The number of seconds we start sleeping for.
- **interval\_step** – How many seconds added to the interval for each retry.
- **interval\_max** – Maximum number of seconds to sleep between each retry.

**Example**

This is an example ensuring a publish operation:

```
>>> from kombu import Connection, Producer
>>> conn = Connection('amqp://')
>>> producer = Producer(conn)

>>> def errback(exc, interval):
...     logger.error('Error: %r', exc, exc_info=1)
...     logger.info('Retry in %s seconds.', interval)

>>> publish = conn.ensure(producer, producer.publish,
...                       errback=errback, max_retries=3)
>>> publish({'hello': 'world'}, routing_key='dest')
```

**ensure\_connection** (*errback=None, max\_retries=None, interval\_start=2, interval\_step=2, interval\_max=30, callback=None*)

Ensure we have a connection to the server.

If not retry establishing the connection with the settings specified.

#### Parameters

- **errback** – Optional callback called each time the connection can't be established. Arguments provided are the exception raised and the interval that will be slept (*exc, interval*).
- **max\_retries** – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval\_start** – The number of seconds we start sleeping for.
- **interval\_step** – How many seconds added to the interval for each retry.
- **interval\_max** – Maximum number of seconds to sleep between each retry.
- **callback** – Optional callback that is called for every internal iteration (1 s)

**failover\_strategies** = {'round-robin': <type 'itertools.cycle'>, 'shuffle': <function shufflecycle at 0x7fb74808d050>}

Map of failover strategy name to Callable

**failover\_strategy** = 'round-robin'

Strategy used to select new hosts when reconnecting after connection failure. One of "round-robin", "shuffle" or any custom iterator constantly yielding new URLs to try.

**get\_heartbeat\_interval** ()

**get\_manager** (*\*args, \*\*kwargs*)

**get\_transport\_cls** ()

Get the currently used transport class.

**heartbeat** = None

Heartbeat value, currently only supported by the py-amqp transport.

**heartbeat\_check** (*rate=2*)

Allow the transport to perform any periodic tasks required to make heartbeats work. This should be called approximately every second.

If the current transport does not support heartbeats then this is a noop operation.

**Parameters rate** – Rate is how often the tick is called compared to the actual heartbeat value.

E.g. if the heartbeat is set to 3 seconds, and the tick is called every 3 / 2 seconds, then the rate is 2. This value is currently unused by any transports.

**host**

The host as a host name/port pair separated by colon.

**hostname** = None

**info** ()

Get connection info.

**is\_evented**

**login\_method** = None

**manager**

Experimental manager that can be used to manage/monitor the broker instance. Not available for all transports.

**maybe\_close\_channel** (*channel*)

Close given channel, but ignore connection and channel errors.

**maybe\_switch\_next** ()

Switch to next URL given by the current failover strategy (if any).

**password** = None

**port** = None

**qos\_semantics\_matches\_spec**

**recoverable\_channel\_errors**

List of channel related exceptions that can be automatically recovered from without re-establishing the connection.

**recoverable\_connection\_errors**

List of connection related exceptions that can be recovered from, but where the connection must be closed and re-established first.

**register\_with\_event\_loop** (*loop*)

**release** ()

Close the connection (if open).

**revive** (*new\_channel*)

Revive connection after connection re-established.

**ssl** = None

**supports\_heartbeats**

**switch** (*url*)

Switch connection parameters to use a new URL (does not reconnect)

**transport**

**transport\_options** = None

Additional transport specific options, passed on to the transport instance.

**uri\_prefix** = None

**userid** = None

**virtual\_host** = '/'

## 4.14.2 Pools

See also:

The shortcut methods `Connection.Pool()` and `Connection.ChannelPool()` is the recommended way to instantiate these classes.

`class kombu.connection.ConnectionPool (connection, limit=None, preload=None)`

`LimitExceeded = <class 'kombu.exceptions.ConnectionLimitExceeded'>`

`acquire (block=False, timeout=None)`

Acquire resource.

**Parameters**

- **block** – If the limit is exceeded, block until there is an available item.

- **timeout** – Timeout to wait if `block` is true. Default is `None` (forever).

**Raises `LimitExceeded`** – if `block` is false and the limit has been exceeded.

**release** (*resource*)

**force\_close\_all** ()

Close and remove all resources in the pool (also those in use).

Can be used to close resources from parent processes after fork (e.g. sockets/connections).

**class** `kombu.connection.ChannelPool` (*connection, limit=None, preload=None*)

**LimitExceeded** = <class ‘`kombu.exceptions.ChannelLimitExceeded`’>

**acquire** (*block=False, timeout=None*)

Acquire resource.

**Parameters**

- **block** – If the limit is exceeded, block until there is an available item.
- **timeout** – Timeout to wait if `block` is true. Default is `None` (forever).

**Raises `LimitExceeded`** – if `block` is false and the limit has been exceeded.

**release** (*resource*)

**force\_close\_all** ()

Close and remove all resources in the pool (also those in use).

Can be used to close resources from parent processes after fork (e.g. sockets/connections).

## 4.15 Message Objects - kombu.message

- `kombu.transport.message`

### 4.15.1 kombu.transport.message

Message class.

**class** `kombu.message.Message` (*channel, body=None, delivery\_tag=None, content\_type=None, content\_encoding=None, delivery\_info={}, properties=None, headers=None, postencode=None, accept=None, \*\*kwargs*)

Base class for received messages.

**exception `MessageStateError`**

The message has already been acknowledged.

`Message.accept`

`Message.ack` ()

Acknowledge this message as being processed., This will remove the message from the queue.

**Raises `MessageStateError`** – If the message has already been acknowledged/requeued/rejected.

`Message.ack_log_error` (*logger, errors*)

Message.**acknowledged**

Set to true if the message has been acknowledged.

Message.**body**

Message.**channel**

Message.**content\_encoding**

Message.**content\_type**

Message.**decode** ()

Deserialize the message body, returning the original python structure sent by the publisher.

Message.**delivery\_info**

Message.**delivery\_tag**

Message.**errors** = None

Message.**headers**

Message.**payload**

The decoded message body.

Message.**properties**

Message.**reject** (*requeue=False*)

Reject this message.

The message will be discarded by the server.

**Raises MessageStateError** – If the message has already been acknowledged/requeued/rejected.

Message.**reject\_log\_error** (*logger, errors, requeue=False*)

Message.**requeue** ()

Reject this message and put it back on the queue.

You must not use this method as a means of selecting messages to process.

**Raises MessageStateError** – If the message has already been acknowledged/requeued/rejected.

## 4.16 kombu.compression

Compression utilities.

- *Encoding/decoding*
- *Registry*

### 4.16.1 Encoding/decoding

kombu.compression.**compress** (*body, content\_type*)

Compress text.

**Parameters**

- **body** – The text to compress.



- **content\_type** – mime-type of compression method to use.

`kombu.compression.decompress` (*body*, *content\_type*)

Decompress compressed text.

#### Parameters

- **body** – Previously compressed text to uncompress.
- **content\_type** – mime-type of compression method used.

## 4.16.2 Registry

`kombu.compression.encoders` ()

Return a list of available compression methods.

`kombu.compression.get_encoder` (*t*)

Get encoder by alias name.

`kombu.compression.get_decoder` (*t*)

Get decoder by alias name.

`kombu.compression.register` (*encoder*, *decoder*, *content\_type*, *aliases=[]*)

Register new compression method.

#### Parameters

- **encoder** – Function used to compress text.
- **decoder** – Function used to decompress previously compressed text.
- **content\_type** – The mime type this compression method identifies as.
- **aliases** – A list of names to associate with this compression method.

## 4.17 General Pools - kombu.pools

- *kombu.pools*

### 4.17.1 kombu.pools

Public resource pools.

`class kombu.pools.ProducerPool` (*connections*, *\*args*, *\*\*kwargs*)

`class Producer` (*channel*, *exchange=None*, *routing\_key=None*, *serializer=None*, *auto\_declare=None*,  
*compression=None*, *on\_return=None*)

Message Producer.

#### Parameters

- **channel** – Connection or channel.
- **exchange** – Optional default exchange.
- **routing\_key** – Optional default routing key.

- **serializer** – Default serializer. Default is “*json*”.
- **compression** – Default compression method. Default is no compression.
- **auto\_declare** – Automatically declare the default exchange at instantiation. Default is `True`.
- **on\_return** – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to `publish()` is used. This callback needs the following signature: (*exception, exchange, routing\_key, message*). Note that the producer needs to drain events to use this feature.

**auto\_declare = True**

**channel**

**close()**

**compression = None**

**connection**

**declare()**

Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

**exchange = None**

**maybe\_declare** (*entity, retry=False, \*\*retry\_policy*)

Declare the exchange if it hasn't already been declared during this session.

**on\_return = None**

**publish** (*body, routing\_key=None, delivery\_mode=None, mandatory=False, immediate=False, priority=0, content\_type=None, content\_encoding=None, serializer=None, headers=None, compression=None, exchange=None, retry=False, retry\_policy=None, declare=[], expiration=None, \*\*properties*)

Publish message to the specified exchange.

**Parameters**

- **body** – Message body.
- **routing\_key** – Message routing key.
- **delivery\_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content\_type** – Content type. Default is auto-detect.
- **content\_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry\_policy** – Retry configuration, this is the keywords supported by `ensure()`.
- **expiration** – A TTL in seconds can be specified per message. Default is no expiration.
- **\*\*properties** – Additional message properties, see AMQP spec.

**release()**

```

    revive (channel)
        Revive the producer after connection loss.

    routing_key = ""

    serializer = None

    ProducerPool.close_resource (resource)

    ProducerPool.create_producer ()

    ProducerPool.new ()

    ProducerPool.prepare (p)

    ProducerPool.release (resource)

    ProducerPool.setup ()

class kombu.pools.PoolGroup (limit=None)

    create (resource, limit)

    kombu.pools.register_group (group)

    kombu.pools.get_limit ()

    kombu.pools.set_limit (limit, force=False, reset_after=False)

    kombu.pools.reset (*args, **kwargs)

```

## 4.18 kombu.abstract

Object utilities.

```

class kombu.abstract.MaybeChannelBound (*args, **kwargs)
    Mixin for classes that can be bound to an AMQP channel.

    bind (channel)
        Create copy of the instance that is bound to a channel.

    can_cache_declaration = False
        Defines whether maybe_declare can skip declaring this entity twice.

    channel
        Current channel if the object is bound.

    is_bound
        Flag set if the channel is bound.

    maybe_bind (channel)
        Bind instance to channel if not already bound.

    revive (channel)
        Revive channel after the connection has been re-established.

        Used by ensure ().

    when_bound ()
        Callback called when the class is bound.

```

## 4.19 Async Utilities - kombu.syn

- *kombu.syn*

### 4.19.1 kombu.syn

`kombu.syn.detect_environment()`

## 4.20 Event Loop - kombu.async

- *kombu.async*

### 4.20.1 kombu.async

Event loop implementation.

**class** `kombu.async.Hub` (*timer=None*)  
Event loop object.

**Parameters** `timer` – Specify timer object.

**ERR = 24**

**READ = 1**

**WRITE = 4**

**add** (*fd, callback, flags, args=(), consolidate=False*)

**add\_reader** (*fds, callback, \*args*)

**add\_writer** (*fds, callback, \*args*)

**call\_at** (*when, callback, \*args*)

**call\_later** (*delay, callback, \*args*)

**call\_repeatedly** (*delay, callback, \*args*)

**call\_soon** (*callback, \*args*)

**close** (*\*args*)

**create\_loop** (*generator=<type 'generator'>, sleep=<built-in function sleep>, min=<built-in function min>, next=<built-in function next>, Empty=<class 'Queue.Empty'>, StopIteration=<type 'exceptions.StopIteration'>, KeyError=<type 'exceptions.KeyError'>, READ=1, WRITE=4, ERR=24*)

**fire\_timers** (*min\_delay=1, max\_delay=10, max\_timers=10, propagate=()*)

**loop**

**on\_close** = None

**remove** (*fd*)

```

remove_reader(fd)
remove_writer(fd)
repr_active()
repr_events(events)
reset()
run_forever()
run_once()
scheduler
stop()

```

```
kombu.async.get_event_loop()
```

```
kombu.async.set_event_loop(loop)
```

## 4.21 Event Loop Implementation - kombu.async.hub

- *kombu.async.hub*

### 4.21.1 kombu.async.hub

Event loop implementation.

```
class kombu.async.hub.Hub(timer=None)
```

Event loop object.

**Parameters** *timer* – Specify timer object.

**ERR = 24**

Flag set on error, and the fd should be read from asap.

**READ = 1**

Flag set if reading from an fd will not block.

**WRITE = 4**

Flag set if writing to an fd will not block.

```
add(fd, callback, flags, args=(), consolidate=False)
```

```
add_reader(fds, callback, *args)
```

```
add_writer(fds, callback, *args)
```

```
call_at(when, callback, *args)
```

```
call_later(delay, callback, *args)
```

```
call_repeatedly(delay, callback, *args)
```

```
call_soon(callback, *args)
```

```
close(*args)
```

```
create_loop (generator=<type 'generator'>, sleep=<built-in function sleep>, min=<built-in function min>, next=<built-in function next>, Empty=<class 'Queue.Empty'>, StopIteration=<type 'exceptions.StopIteration'>, KeyError=<type 'exceptions.KeyError'>, READ=1, WRITE=4, ERR=24)
```

```
fire_timers (min_delay=1, max_delay=10, max_timers=10, propagate=())
```

```
loop
```

```
on_close = None
```

List of callbacks to be called when the loop is exiting, applied with the hub instance as sole argument.

```
remove (fd)
```

```
remove_reader (fd)
```

```
remove_writer (fd)
```

```
repr_active ()
```

```
repr_events (events)
```

```
reset ()
```

```
run_forever ()
```

```
run_once ()
```

```
scheduler
```

```
stop ()
```

```
kombu.async.hub.get_event_loop ()
```

```
kombu.async.hub.set_event_loop (loop)
```

## 4.22 Semaphores - kombu.async.semaphore

- [kombu.async.semaphore](#)

### 4.22.1 kombu.async.semaphore

Semaphores and concurrency primitives.

```
class kombu.async.semaphore.DummyLock
```

Pretending to be a lock.

```
class kombu.async.semaphore.LaxBoundedSemaphore (value)
```

Asynchronous Bounded Semaphore.

Lax means that the value will stay within the specified range even if released more times than it was acquired.

Example:

```
>>> from future import print_statement as printf
# ^ ignore: just fooling stupid pyflakes
```

```
>>> x = LaxBoundedSemaphore (2)
```

```
>>> x.acquire(printf, 'HELLO 1')
HELLO 1
```

```
>>> x.acquire(printf, 'HELLO 2')
HELLO 2
```

```
>>> x.acquire(printf, 'HELLO 3')
>>> x._waiters # private, do not access directly
[print, ('HELLO 3', )]
```

```
>>> x.release()
HELLO 3
```

**acquire** (*callback*, *\*partial\_args*)

Acquire semaphore, applying callback if the resource is available.

**Parameters**

- **callback** – The callback to apply.
- **\*partial\_args** – partial arguments to callback.

**clear** ()

Reset the semaphore, which also wipes out any waiting callbacks.

**grow** (*n=1*)

Change the size of the semaphore to accept more users.

**release** ()

Release semaphore.

If there are any waiters this will apply the first waiter that is waiting for the resource (FIFO order).

**shrink** (*n=1*)

Change the size of the semaphore to accept less users.

## 4.23 Timer - kombu.async.timer

- *kombu.async.timer*

### 4.23.1 kombu.async.timer

Timer scheduling Python callbacks.

**class** kombu.async.timer.**Entry** (*fun*, *args=None*, *kwargs=None*)

**args**

**cancel** ()

**cancelled**

**fun**

**kwargs**

**tref**

**class** kombu.async.timer.**Timer** (*max\_interval=None, on\_error=None, \*\*kwargs*)  
ETA scheduler.

**class** **Entry** (*fun, args=None, kwargs=None*)

**args**

**cancel** ()

**cancelled**

**fun**

**kwargs**

**tref**

**Timer.apply\_entry** (*entry*)

**Timer.call\_after** (*secs, fun, args=(), kwargs={}, priority=0*)

**Timer.call\_at** (*eta, fun, args=(), kwargs={}, priority=0*)

**Timer.call\_repeatedly** (*secs, fun, args=(), kwargs={}, priority=0*)

**Timer.cancel** (*tref*)

**Timer.clear** ()

**Timer.enter\_after** (*secs, entry, priority=0, time=<built-in function time>*)

**Timer.enter\_at** (*entry, eta=None, priority=0, time=<built-in function time>*)  
Enter function into the scheduler.

#### Parameters

- **entry** – Item to enter.
- **eta** – Scheduled time as a `datetime.datetime` object.
- **priority** – Unused.

**Timer.handle\_error** (*exc\_info*)

**Timer.on\_error** = **None**

**Timer.queue**

Snapshot of underlying datastructure.

**Timer.schedule**

**Timer.stop** ()

**kombu.async.timer.to\_timestamp** (*d, default\_timezone=<UTC>*)

## 4.24 Debugging Utils - kombu.async.debug

**kombu.async.debug.callback\_for** (*h, fd, flag, \*default*)

**kombu.async.debug.repr\_active** (*h*)

**kombu.async.debug.repr\_events** (*h, events*)

**kombu.async.debug.repr\_flag** (*flag*)



`kombu.async.debug.repr_readers` (*h*)

`kombu.async.debug.repr_writers` (*h*)

## 4.25 kombu.transport

Built-in transports.

- *Data*
- *Functions*

### 4.25.1 Data

`kombu.transport.DEFAULT_TRANSPORT`

Default transport used when no transport specified.

`kombu.transport.TRANSPORT_ALIASES`

Mapping of transport aliases/class names.

### 4.25.2 Functions

`kombu.transport.get_transport_cls` (*transport=None*)

Get transport class by name.

The transport string is the full path to a transport class, e.g.:

```
"kombu.transport.pyamqp:Transport"
```

If the name does not include "." (is not fully qualified), the alias table will be consulted.

`kombu.transport.resolve_transport` (*transport=None*)

## 4.26 kombu.transport.pyamqp

pure python amqp transport.

- *Transport*
- *Connection*
- *Channel*
- *Message*

### 4.26.1 Transport

```
class kombu.transport.pyamqp.Transport(client, default_port=None, default_ssl_port=None,
                                       **kwargs)
```

```
class Connection (host='localhost', userid='guest', password='guest', login_method='AMQPPLAIN',
login_response=None, virtual_host='/', locale='en_US', client_properties=None,
ssl=False, connect_timeout=None, channel_max=None, frame_max=None,
heartbeat=0, on_blocked=None, on_unblocked=None, confirm_publish=False,
**kwargs)
```

```
class Channel (connection, channel_id=None, auto_decode=True)
```

```
class Message (channel, msg, **kwargs)
```

```
Transport.Connection.Channel.message_to_python (raw_message)
```

Convert encoded message body back to a Python value.

```
Transport.Connection.Channel.prepare_message (body, priority=None,
content_type=None, content_encoding=None, headers=None, properties=None,
_Message=<class 'amqp.basic_message.Message'>)
```

Prepares message so that it can be sent using this transport.

```
Transport.channel_errors = (<class 'amqp.exceptions.ChannelError'>,)
```

```
Transport.close_connection (connection)
```

Close the AMQP broker connection.

```
Transport.connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'socket.error'>, <type 'excepti
```

```
Transport.create_channel (connection)
```

```
Transport.default_connection_params
```

```
Transport.default_port = 5672
```

```
Transport.default_ssl_port = 5671
```

```
Transport.drain_events (connection, **kwargs)
```

```
Transport.driver_name = 'py-amqp'
```

```
Transport.driver_type = 'amqp'
```

```
Transport.driver_version ()
```

```
Transport.establish_connection ()
```

Establish connection to the AMQP broker.

```
Transport.get_heartbeat_interval (connection)
```

```
Transport.get_manager (*args, **kwargs)
```

```
Transport.heartbeat_check (connection, rate=2)
```

```
Transport.qos_semantics_matches_spec (connection)
```

```
Transport.recoverable_channel_errors = (<class 'amqp.exceptions.RecoverableChannelError'>,)
```

```
Transport.recoverable_connection_errors = (<class 'amqp.exceptions.RecoverableConnectionError'>, <class
```

```
Transport.register_with_event_loop (connection, loop)
```

```
Transport.supports_ev = True
```

```
Transport.supports_heartbeats = True
```

```
Transport.verify_connection (connection)
```

## 4.26.2 Connection

```
class kombu.transport.pyamqp.Connection (host='localhost',      userid='guest',      pass-
                                         word='guest',      login_method='AMQPLAIN',
                                         login_response=None, virtual_host='/', lo-
                                         cale='en_US', client_properties=None, ssl=False,
                                         connect_timeout=None, channel_max=None,
                                         frame_max=None, heartbeat=0, on_blocked=None,
                                         on_unblocked=None, confirm_publish=False,
                                         **kwargs)
```

```
class Channel (connection, channel_id=None, auto_decode=True)
```

```
Consumer (*args, **kwargs)
```

```
class Message (channel, msg, **kwargs)
```

```
exception MessageStateError
```

The message has already been acknowledged.

**args**

**message**

```
Connection.Channel.Message.accept
```

```
Connection.Channel.Message.ack ()
```

Acknowledge this message as being processed., This will remove the message from the queue.

**Raises MessageStateError** – If the message has already been acknowl-  
edged/requeued/rejected.

```
Connection.Channel.Message.ack_log_error (logger, errors)
```

```
Connection.Channel.Message.acknowledged
```

Set to true if the message has been acknowledged.

```
Connection.Channel.Message.body
```

```
Connection.Channel.Message.channel
```

```
Connection.Channel.Message.content_encoding
```

```
Connection.Channel.Message.content_type
```

```
Connection.Channel.Message.decode ()
```

Deserialize the message body, returning the original python structure sent by the publisher.

```
Connection.Channel.Message.delivery_info
```

```
Connection.Channel.Message.delivery_tag
```

```
Connection.Channel.Message.errors = None
```

```
Connection.Channel.Message.headers
```

```
Connection.Channel.Message.payload
```

The decoded message body.

```
Connection.Channel.Message.properties
```

```
Connection.Channel.Message.reject (requeue=False)
```

Reject this message.

The message will be discarded by the server.

**Raises `MessageStateError`** – If the message has already been acknowledged/requeued/rejected.

`Connection.Channel.Message.reject_log_error` (*logger*, *errors*, *re-queue=False*)

`Connection.Channel.Message.requeue` ()

Reject this message and put it back on the queue.

You must not use this method as a means of selecting messages to process.

**Raises `MessageStateError`** – If the message has already been acknowledged/requeued/rejected.

`Connection.Channel.Producer` (*\*args*, *\*\*kwargs*)

`Connection.Channel.after_reply_message_received` (*queue*)

reply queue semantics: can be used to delete the queue after transient reply message received.

`Connection.Channel.basic_ack` (*delivery\_tag*, *multiple=False*)

Acknowledge one or more messages

This method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. The client can ask to confirm a single message or a set of messages up to and including a specific message.

**PARAMETERS:** *delivery\_tag*: longlong  
server-assigned delivery tag

The server-assigned and channel-specific delivery tag

**RULE:**

The delivery tag is valid only within the channel from which the message was received. I.e. a client MUST NOT receive a message on one channel and then acknowledge it on another.

**RULE:**

The server MUST NOT use a zero value for delivery tags. Zero is reserved for client use, meaning “all messages so far received”.

*multiple*: boolean

acknowledge multiple messages

If set to True, the delivery tag is treated as “up to and including”, so that the client can acknowledge multiple messages with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is True, and the delivery tag is zero, tells the server to acknowledge all outstanding messages.

**RULE:**

The server MUST validate that a non-zero delivery- tag refers to an delivered message, and raise a channel exception if this is not the case.

`Connection.Channel.basic_cancel` (*consumer\_tag*, *nowait=False*)

End a queue consumer

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer. The client may receive an arbitrary number of messages in between sending the cancel method and receiving the cancel-ok reply.

**RULE:**

If the queue no longer exists when the client sends a cancel command, or the consumer has been cancelled for other reasons, this command has no effect.

**PARAMETERS:** *consumer\_tag*: shortstr

consumer tag

Identifier for the consumer, valid within the current connection.

**RULE:**

The consumer tag is valid only within the channel from which the consumer was created. I.e. a client **MUST NOT** create a consumer in one channel and then use it in another.

nowait: boolean

do not send a reply method

If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

```
Connection.Channel.basic_consume(queue='', consumer_tag='', no_local=False,
                                   no_ack=False, exclusive=False, nowait=False,
                                   callback=None, arguments=None,
                                   on_cancel=None)
```

Start a queue consumer

This method asks the server to start a “consumer”, which is a transient request for messages from a specific queue. Consumers last as long as the channel they were created on, or until the client cancels them.

**RULE:**

The server **SHOULD** support at least 16 consumers per queue, unless the queue was declared as private, and ideally, impose no limit except as defined by available resources.

**PARAMETERS:** queue: shortstr

Specifies the name of the queue to consume from. If the queue name is null, refers to the current queue for the channel, which is the last declared queue.

**RULE:**

If the client did not previously declare a queue, and the queue name in this method is empty, the server **MUST** raise a connection exception with reply code 530 (not allowed).

consumer\_tag: shortstr

Specifies the identifier for the consumer. The consumer tag is local to a connection, so two clients can use the same consumer tags. If this field is empty the server will generate a unique tag.

**RULE:**

The tag **MUST NOT** refer to an existing consumer. If the client attempts to create two consumers with the same non-empty tag the server **MUST** raise a connection exception with reply code 530 (not allowed).

no\_local: boolean

do not deliver own messages

If the no-local field is set the server will not send messages to the client that published them.

no\_ack: boolean

no acknowledgement needed

If this field is set the server does not expect acknowledgments for messages. That is, when a message is delivered to the client the server automatically and silently acknowledges it on behalf of the client. This functionality increases performance but at the cost of reliability. Messages can get lost if a client dies before it can deliver them to the application.

exclusive: boolean

request exclusive access

Request exclusive consumer access, meaning only this consumer can access the queue.

**RULE:**

If the server cannot grant exclusive access to the queue when asked, - because there are other consumers active - it **MUST** raise a channel exception with return code 403 (access refused).

`nowait`: boolean

do not send a reply method

If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

`callback`: Python callable

function/method called with each delivered message

For each message delivered by the broker, the callable will be called with a Message object as the single argument. If no callable is specified, messages are quietly discarded, `no_ack` should probably be set to True in that case.

`Connection.Channel.basic_get` (*queue*='', *no\_ack=False*)

Direct access to a queue

This method provides a direct access to the messages in a queue using a synchronous dialogue that is designed for specific types of application where synchronous functionality is more important than performance.

**PARAMETERS:** `queue`: shortstr

Specifies the name of the queue to consume from. If the queue name is null, refers to the current queue for the channel, which is the last declared queue.

**RULE:**

If the client did not previously declare a queue, and the queue name in this method is empty, the server **MUST** raise a connection exception with reply code 530 (not allowed).

`no_ack`: boolean

no acknowledgement needed

If this field is set the server does not expect acknowledgments for messages. That is, when a message is delivered to the client the server automatically and silently acknowledges it on behalf of the client. This functionality increases performance but at the cost of reliability. Messages can get lost if a client dies before it can deliver them to the application.

Non-blocking, returns a message object, or None.

`Connection.Channel.basic_publish` (*msg*, *exchange*='', *routing\_key*='', *mandatory=False*, *immediate=False*)

Publish a message

This method publishes a message to a specific exchange. The message will be routed to queues as defined by the exchange configuration and distributed to any active consumers when the transaction, if any, is committed.

**PARAMETERS:** `exchange`: shortstr

Specifies the name of the exchange to publish to. The exchange name can be empty, meaning the default exchange. If the exchange name is specified, and that exchange does not exist, the server will raise a channel exception.

**RULE:**

The server **MUST** accept a blank exchange name to mean the default exchange.

**RULE:**

The exchange **MAY** refuse basic content in which case it **MUST** raise a channel exception with reply code 540 (not implemented).

`routing_key`: shortstr  
 Message routing key

Specifies the routing key for the message. The routing key is used for routing messages depending on the exchange configuration.

`mandatory`: boolean  
 indicate mandatory routing

This flag tells the server how to react if the message cannot be routed to a queue. If this flag is True, the server will return an unroutable message with a Return method. If this flag is False, the server silently drops the message.

**RULE:**

The server SHOULD implement the mandatory flag.

`immediate`: boolean  
 request immediate delivery

This flag tells the server how to react if the message cannot be routed to a queue consumer immediately. If this flag is set, the server will return an undeliverable message with a Return method. If this flag is zero, the server will queue the message, but with no guarantee that it will ever be consumed.

**RULE:**

The server SHOULD implement the immediate flag.

`Connection.Channel.basic_publish_confirm` (*\*args, \*\*kwargs*)

`Connection.Channel.basic_qos` (*prefetch\_size, prefetch\_count, a\_global*)  
 Specify quality of service

This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The particular properties and semantics of a qos method always depend on the content class semantics. Though the qos method could in principle apply to both peers, it is currently meaningful only for the server.

**PARAMETERS:** `prefetch_size`: long  
 prefetch window in octets

The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement. This field specifies the prefetch window size in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls into other prefetch limits). May be set to zero, meaning “no specific limit”, although other prefetch limits may still apply. The prefetch-size is ignored if the no-ack option is set.

**RULE:**

The server MUST ignore this setting when the client is not processing any messages - i.e. the prefetch size does not limit the transfer of single messages to a client, only the sending in advance of more messages while the client still has one or more unacknowledged messages.

`prefetch_count`: short  
 prefetch window in messages

Specifies a prefetch window in terms of whole messages. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both prefetch windows (and those at the channel and connection level) allow it. The prefetch-count is ignored if the no-ack option is set.

**RULE:**

The server MAY send less data in advance than allowed by the client's specified prefetch windows but it MUST NOT send more.

`a_global`: boolean  
apply to entire connection

By default the QoS settings apply to the current channel only. If this field is set, they are applied to the entire connection.

`Connection.Channel.basic_recover` (*requeue=False*)

Redeliver unacknowledged messages

This method asks the broker to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method is only allowed on non-transacted channels.

RULE:

The server MUST set the redelivered flag on all messages that are resent.

RULE:

The server MUST raise a channel exception if this is called on a transacted channel.

**PARAMETERS:** `requeue`: boolean  
requeue the message

If this field is False, the message will be redelivered to the original recipient. If this field is True, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

`Connection.Channel.basic_recover_async` (*requeue=False*)

`Connection.Channel.basic_reject` (*delivery\_tag, requeue*)

Reject an incoming message

This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

RULE:

The server SHOULD be capable of accepting and process the Reject method while sending message content with a Deliver or Get-Ok method. I.e. the server should read and process incoming methods while sending output frames. To cancel a partially-send content, the server sends a content body frame of size 1 (i.e. with no data except the frame-end octet).

RULE:

The server SHOULD interpret this method as meaning that the client is unable to process the message at this time.

RULE:

A client MUST NOT use this method as a means of selecting messages to process. A rejected message MAY be discarded or dead-lettered, not necessarily passed to another client.

**PARAMETERS:** `delivery_tag`: longlong  
server-assigned delivery tag

The server-assigned and channel-specific delivery tag

RULE:

The delivery tag is valid only within the channel from which the message was received. I.e. a client MUST NOT receive a message on one channel and then acknowledge it on another.

RULE:

The server MUST NOT use a zero value for delivery tags. Zero is reserved for client use, meaning "all messages so far received".

`requeue`: boolean  
requeue the message



If this field is False, the message will be discarded. If this field is True, the server will attempt to requeue the message.

**RULE:**

The server **MUST NOT** deliver the message to the same client within the context of the current channel. The recommended strategy is to attempt to deliver the message to an alternative consumer, and if that is not possible, to move the message to a dead-letter queue. The server **MAY** use more sophisticated tracking to hold the message on the queue and redeliver it to the same client at a later stage.

`Connection.Channel.close` (*reply\_code=0, reply\_text='', method\_sig=(0, 0)*)

Request a channel close

This method indicates that the sender wants to close the channel. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class and method id of the method which caused the exception.

**RULE:**

After sending this method any received method except Channel.Close-OK **MUST** be discarded.

**RULE:**

The peer sending this method **MAY** use a counter or timeout to detect failure of the other peer to respond correctly with Channel.Close-OK..

**PARAMETERS:** `reply_code`: short

The reply code. The AMQ reply codes are defined in AMQ RFC 011.

`reply_text`: shortstr

The localised reply text. This text can be logged as an aid to resolving issues.

`class_id`: short

failing method class

When the close is provoked by a method exception, this is the class of the method.

`method_id`: short

failing method ID

When the close is provoked by a method exception, this is the ID of the method.

`Connection.Channel.confirm_select` (*nowait=False*)

Enables publisher confirms for this channel (an RabbitMQ extension).

Can now be used if the channel is in transactional mode.

**Parameters** `nowait` – If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

`Connection.Channel.dispatch_method` (*method\_sig, args, content*)

`Connection.Channel.exchange_bind` (*destination, source='', routing\_key='', nowait=False, arguments=None*)

This method binds an exchange to an exchange.

**RULE:**

A server **MUST** allow and ignore duplicate bindings - that is, two or more bind methods for a specific exchanges, with identical arguments - without treating these as an error.

**RULE:**

A server **MUST** allow cycles of exchange bindings to be created including allowing an exchange to be bound to itself.

**RULE:**

A server **MUST** not deliver the same message more than once to a destination exchange, even if the topology of exchanges and bindings results in multiple (even infinite) routes to that exchange.

**PARAMETERS:** reserved-1: short

destination: shortstr

Specifies the name of the destination exchange to bind.

**RULE:**

A client **MUST NOT** be allowed to bind a non- existent destination exchange.

**RULE:**

The server **MUST** accept a blank exchange name to mean the default exchange.

source: shortstr

Specifies the name of the source exchange to bind.

**RULE:**

A client **MUST NOT** be allowed to bind a non- existent source exchange.

**RULE:**

The server **MUST** accept a blank exchange name to mean the default exchange.

routing-key: shortstr

Specifies the routing key for the binding. The routing key is used for routing messages depending on the exchange configuration. Not all exchanges use a routing key - refer to the specific exchange documentation.

no-wait: bit

arguments: table

A set of arguments for the binding. The syntax and semantics of these arguments depends on the exchange class.

`Connection.Channel.exchange_declare` (*exchange, type, passive=False, durable=False, auto\_delete=True, nowait=False, arguments=None*)

Declare exchange, create if needed

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

**RULE:**

The server **SHOULD** support a minimum of 16 exchanges per virtual host and ideally, impose no limit except as defined by available resources.

**PARAMETERS:** exchange: shortstr

**RULE:**

Exchange names starting with “amq.” are reserved for predeclared and standardised exchanges. If the client attempts to create an exchange starting with “amq.”, the server **MUST** raise a channel exception with reply code 403 (access refused).

type: shortstr

exchange type

Each exchange belongs to one of a set of exchange types implemented by the server. The exchange types define the functionality of the exchange - i.e. how messages are routed through it. It is not valid or meaningful to attempt to change the type of an existing exchange.

**RULE:**

If the exchange already exists with a different type, the server **MUST** raise a connection exception with a reply code 507 (not allowed).

**RULE:**

If the server does not support the requested exchange type it **MUST** raise a connection exception with a reply code 503 (command invalid).

passive: boolean

do not create exchange

If set, the server will not create the exchange. The client can use this to check whether an exchange exists without modifying the server state.

RULE:

If set, and the exchange does not already exist, the server **MUST** raise a channel exception with reply code 404 (not found).

durable: boolean

request a durable exchange

If set when creating a new exchange, the exchange will be marked as durable. Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged if/when a server restarts.

RULE:

The server **MUST** support both durable and transient exchanges.

RULE:

The server **MUST** ignore the durable field if the exchange already exists.

auto\_delete: boolean

auto-delete when unused

If set, the exchange is deleted when all queues have finished using it.

RULE:

The server **SHOULD** allow for a reasonable delay between the point when it determines that an exchange is not being used (or no longer used), and the point when it deletes the exchange. At the least it must allow a client to create an exchange and then bind a queue to it, with a small but non-zero delay between these two actions.

RULE:

The server **MUST** ignore the auto-delete field if the exchange already exists.

nowait: boolean

do not send a reply method

If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

arguments: table

arguments for declaration

A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation. This field is ignored if passive is True.

`Connection.Channel.exchange_delete` (*exchange*, *if\_unused=False*, *nowait=False*)

Delete an exchange

This method deletes an exchange. When an exchange is deleted all queue bindings on the exchange are cancelled.

**PARAMETERS:** exchange: shortstr

RULE:

The exchange **MUST** exist. Attempting to delete a non-existing exchange causes a channel exception.

if\_unused: boolean

delete only if unused

If set, the server will only delete the exchange if it has no queue bindings. If the exchange has queue bindings the server does not delete it but raises a channel exception instead.

RULE:

If set, the server SHOULD delete the exchange but only if it has no queue bindings.

RULE:

If set, the server SHOULD raise a channel exception if the exchange is in use.

nowait: boolean

do not send a reply method

If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

`Connection.Channel.exchange_unbind(destination, source='', routing_key='',  
nowait=False, arguments=None)`

This method unbinds an exchange from an exchange.

RULE:

If a unbind fails, the server MUST raise a connection exception.

**PARAMETERS:** reserved-1: short

destination: shortstr

Specifies the name of the destination exchange to unbind.

RULE:

The client MUST NOT attempt to unbind an exchange that does not exist from an exchange.

RULE:

The server MUST accept a blank exchange name to mean the default exchange.

source: shortstr

Specifies the name of the source exchange to unbind.

RULE:

The client MUST NOT attempt to unbind an exchange from an exchange that does not exist.

RULE:

The server MUST accept a blank exchange name to mean the default exchange.

routing-key: shortstr

Specifies the routing key of the binding to unbind.

no-wait: bit

arguments: table

Specifies the arguments of the binding to unbind.

`Connection.Channel.flow(active)`

Enable/disable flow from peer

This method asks the peer to pause or restart the flow of content data. This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process. Note that this method is not intended for window control. The peer that receives a request to stop sending content should finish sending the current content, if any, and then wait until it receives a Flow restart method.

RULE:

When a new channel is opened, it is active. Some applications assume that channels are inactive until started. To emulate this behaviour a client MAY open the channel, then pause it.

RULE:

When sending content data in multiple frames, a peer SHOULD monitor the channel for incoming methods and respond to a Channel.Flow as rapidly as possible.

RULE:

A peer MAY use the Channel.Flow method to throttle incoming content data for internal reasons, for example, when exchanging data over a slower connection.

**RULE:**

The peer that requests a Channel.Flow method MAY disconnect and/or ban a peer that does not respect the request.

**PARAMETERS:** active: boolean

start/stop content frames

If True, the peer starts sending content frames. If False, the peer stops sending content frames.

Connection.Channel.get\_bindings()

Connection.Channel.message\_to\_python(raw\_message)

Convert encoded message body back to a Python value.

Connection.Channel.no\_ack\_consumers = None

Connection.Channel.prepare\_message(body, priority=None, content\_type=None, content\_encoding=None, headers=None, properties=None, \_Message=<class 'amqp.basic\_message.Message'>)

Prepares message so that it can be sent using this transport.

Connection.Channel.queue\_bind(queue, exchange='', routing\_key='', nowait=False, arguments=None)

Bind queue to an exchange

This method binds a queue to an exchange. Until a queue is bound it will not receive any messages. In a classic messaging model, store-and-forward queues are bound to a dest exchange and subscription queues are bound to a dest\_wild exchange.

**RULE:**

A server MUST allow ignore duplicate bindings - that is, two or more bind methods for a specific queue, with identical arguments - without treating these as an error.

**RULE:**

If a bind fails, the server MUST raise a connection exception.

**RULE:**

The server MUST NOT allow a durable queue to bind to a transient exchange. If the client attempts this the server MUST raise a channel exception.

**RULE:**

Bindings for durable queues are automatically durable and the server SHOULD restore such bindings after a server restart.

**RULE:**

The server SHOULD support at least 4 bindings per queue, and ideally, impose no limit except as defined by available resources.

**PARAMETERS:** queue: shortstr

Specifies the name of the queue to bind. If the queue name is empty, refers to the current queue for the channel, which is the last declared queue.

**RULE:**

If the client did not previously declare a queue, and the queue name in this method is empty, the server MUST raise a connection exception with reply code 530 (not allowed).

**RULE:**

If the queue does not exist the server MUST raise a channel exception with reply code 404 (not found).

exchange: shortstr

The name of the exchange to bind to.

**RULE:**

If the exchange does not exist the server **MUST** raise a channel exception with reply code 404 (not found).

`routing_key`: shortstr  
message routing key

Specifies the routing key for the binding. The routing key is used for routing messages depending on the exchange configuration. Not all exchanges use a routing key - refer to the specific exchange documentation. If the routing key is empty and the queue name is empty, the routing key will be the current queue for the channel, which is the last declared queue.

`nowait`: boolean  
do not send a reply method

If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

`arguments`: table  
arguments for binding

A set of arguments for the binding. The syntax and semantics of these arguments depends on the exchange class.

```
Connection.Channel.queue_declare(queue='', passive=False, durable=False, exclusive=False, auto_delete=True, nowait=False, arguments=None)
```

Declare queue, create if needed

This method creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

**RULE:**

The server **MUST** create a default binding for a newly- created queue to the default exchange, which is an exchange of type 'direct'.

**RULE:**

The server **SHOULD** support a minimum of 256 queues per virtual host and ideally, impose no limit except as defined by available resources.

**PARAMETERS:** `queue`: shortstr

**RULE:**

The queue name **MAY** be empty, in which case the server **MUST** create a new queue with a unique generated name and return this to the client in the Declare-Ok method.

**RULE:**

Queue names starting with "amq." are reserved for predeclared and standardised server queues. If the queue name starts with "amq." and the passive option is False, the server **MUST** raise a connection exception with reply code 403 (access refused).

`passive`: boolean  
do not create queue

If set, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state.

**RULE:**

If set, and the queue does not already exist, the server **MUST** respond with a reply code 404 (not found) and raise a channel exception.

`durable`: boolean  
request a durable queue

If set when creating a new queue, the queue will be marked as durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue.

RULE:

The server **MUST** recreate the durable queue after a restart.

RULE:

The server **MUST** support both durable and transient queues.

RULE:

The server **MUST** ignore the durable field if the queue already exists.

exclusive: boolean

request an exclusive queue

Exclusive queues may only be consumed from by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'.

RULE:

The server **MUST** support both exclusive (private) and non-exclusive (shared) queues.

RULE:

The server **MUST** raise a channel exception if 'exclusive' is specified and the queue already exists and is owned by a different connection.

auto\_delete: boolean

auto-delete queue when unused

If set, the queue is deleted when all consumers have finished using it. Last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted.

RULE:

The server **SHOULD** allow for a reasonable delay between the point when it determines that a queue is not being used (or no longer used), and the point when it deletes the queue. At the least it must allow a client to create a queue and then create a consumer to read from it, with a small but non-zero delay between these two actions. The server should equally allow for clients that may be disconnected prematurely, and wish to re-consume from the same queue without losing messages. We would recommend a configurable timeout, with a suitable default value being one minute.

RULE:

The server **MUST** ignore the auto-delete field if the queue already exists.

nowait: boolean

do not send a reply method

If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

arguments: table

arguments for declaration

A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation. This field is ignored if passive is True.

**Returns a tuple containing 3 items:** the name of the queue (essential for automatically-named queues) message count consumer count

```
Connection.Channel.queue_delete(queue=' ', if_unused=False, if_empty=False,
                                nowait=False)
```

Delete a queue

This method deletes a queue. When a queue is deleted any pending messages are sent to a dead-letter queue if this is defined in the server configuration, and all consumers on the queue are cancelled.

**RULE:**

The server SHOULD use a dead-letter queue to hold messages that were pending on a deleted queue, and MAY provide facilities for a system administrator to move these messages back to an active queue.

**PARAMETERS:** queue: shortstr

Specifies the name of the queue to delete. If the queue name is empty, refers to the current queue for the channel, which is the last declared queue.

**RULE:**

If the client did not previously declare a queue, and the queue name in this method is empty, the server MUST raise a connection exception with reply code 530 (not allowed).

**RULE:**

The queue must exist. Attempting to delete a non-existing queue causes a channel exception.

if\_unused: boolean

delete only if unused

If set, the server will only delete the queue if it has no consumers. If the queue has consumers the server does not delete it but raises a channel exception instead.

**RULE:**

The server MUST respect the if-unused flag when deleting a queue.

if\_empty: boolean

delete only if empty

If set, the server will only delete the queue if it has no messages. If the queue is not empty the server raises a channel exception.

nowait: boolean

do not send a reply method

If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

`Connection.Channel.queue_purge(queue='', nowait=False)`

Purge a queue

This method removes all messages from a queue. It does not cancel consumers. Purged messages are deleted without any formal “undo” mechanism.

**RULE:**

A call to purge MUST result in an empty queue.

**RULE:**

On transacted channels the server MUST not purge messages that have already been sent to a client but not yet acknowledged.

**RULE:**

The server MAY implement a purge queue or log that allows system administrators to recover accidentally-purged messages. The server SHOULD NOT keep purged messages in the same storage spaces as the live messages since the volumes of purged messages may get very large.

**PARAMETERS:** queue: shortstr

Specifies the name of the queue to purge. If the queue name is empty, refers to the current queue for the channel, which is the last declared queue.

**RULE:**

If the client did not previously declare a queue, and the queue name in this method



is empty, the server MUST raise a connection exception with reply code 530 (not allowed).

**RULE:**

The queue must exist. Attempting to purge a non-existing queue causes a channel exception.

`nowait`: boolean

do not send a reply method

If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

if `nowait` is `False`, returns a `message_count`

`Connection.Channel.queue_unbind` (*queue*, *exchange*, *routing\_key*='', *nowait*=`False`, *arguments*=`None`)

Unbind a queue from an exchange

This method unbinds a queue from an exchange.

**RULE:**

If a unbind fails, the server MUST raise a connection exception.

**PARAMETERS:** `queue`: shortstr

Specifies the name of the queue to unbind.

**RULE:**

The client MUST either specify a queue name or have previously declared a queue on the same channel

**RULE:**

The client MUST NOT attempt to unbind a queue that does not exist.

`exchange`: shortstr

The name of the exchange to unbind from.

**RULE:**

The client MUST NOT attempt to unbind a queue from an exchange that does not exist.

**RULE:**

The server MUST accept a blank exchange name to mean the default exchange.

`routing_key`: shortstr

routing key of binding

Specifies the routing key of the binding to unbind.

`arguments`: table

arguments of binding

Specifies the arguments of the binding to unbind.

`Connection.Channel.tx_commit` ()

Commit the current transaction

This method commits all messages published and acknowledged in the current transaction. A new transaction starts immediately after a commit.

`Connection.Channel.tx_rollback` ()

Abandon the current transaction

This method abandons all messages published and acknowledged in the current transaction. A new transaction starts immediately after a rollback.

`Connection.Channel.tx_select` ()

Select standard transaction mode

This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

`Connection.Channel.wait` (*allowed\_methods=None, timeout=None*)

Wait for a method that matches our `allowed_methods` parameter (the default value of `None` means match any method), and dispatch to it.

`Connection.Transport` (*host, connect\_timeout, ssl=False*)

`Connection.channel` (*channel\_id=None*)

Fetch a Channel object identified by the numeric `channel_id`, or create that object if it doesn't already exist.

`Connection.channel_errors` = (<class 'amqp.exceptions.ChannelError'>,)

`Connection.client_heartbeat` = `None`

`Connection.close` (*reply\_code=0, reply\_text='', method\_sig=(0, 0)*)

Request a connection close

This method indicates that the sender wants to close the connection. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class and method id of the method which caused the exception.

**RULE:**

After sending this method any received method except the Close-OK method MUST be discarded.

**RULE:**

The peer sending this method MAY use a counter or timeout to detect failure of the other peer to respond correctly with the Close-OK method.

**RULE:**

When a server receives the Close method from a client it MUST delete all server-side resources associated with the client's context. A client CANNOT reconnect to a context after sending or receiving a Close method.

**PARAMETERS:** `reply_code`: short

The reply code. The AMQ reply codes are defined in AMQ RFC 011.

`reply_text`: shortstr

The localised reply text. This text can be logged as an aid to resolving issues.

`class_id`: short

failing method class

When the close is provoked by a method exception, this is the class of the method.

`method_id`: short

failing method ID

When the close is provoked by a method exception, this is the ID of the method.

`Connection.connected`

`Connection.connection_errors` = (<class 'amqp.exceptions.ConnectionError'>, <class 'socket.error'>, <type 'exc

`Connection.dispatch_method` (*method\_sig, args, content*)

`Connection.drain_events` (*timeout=None*)

Wait for an event on a channel.

`Connection.heartbeat = None`

`Connection.heartbeat_tick` (*rate=2*)

Send heartbeat packets, if necessary, and fail if none have been received recently. This should be called frequently, on the order of once per second.

**Parameters** `rate` – Ignored

`Connection.is_alive` ()

`Connection.last_heartbeat_received = 0`

`Connection.last_heartbeat_sent = 0`

`Connection.prev_recv = None`

`Connection.prev_sent = None`

`Connection.read_timeout` (*timeout=None*)

`Connection.recoverable_channel_errors = (<class 'amqp.exceptions.RecoverableChannelError'>,)`

`Connection.recoverable_connection_errors = (<class 'amqp.exceptions.RecoverableConnectionError'>, <class 'amqp.exceptions.RecoverableChannelError'>)`

`Connection.send_heartbeat` ()

`Connection.server_capabilities`

`Connection.server_heartbeat = None`

`Connection.sock`

`Connection.wait` (*allowed\_methods=None, timeout=None*)

Wait for a method that matches our `allowed_methods` parameter (the default value of `None` means match any method), and dispatch to it.

### 4.26.3 Channel

`class kombu.transport.pyamqp.Channel` (*connection, channel\_id=None, auto\_decode=True*)

`class Message` (*channel, msg, \*\*kwargs*)

`Channel.message_to_python` (*raw\_message*)

Convert encoded message body back to a Python value.

`Channel.prepare_message` (*body, priority=None, content\_type=None, content\_encoding=None, headers=None, properties=None, \_Message=<class 'amqp.basic\_message.Message'>*)

Prepares message so that it can be sent using this transport.

### 4.26.4 Message

`class kombu.transport.pyamqp.Message` (*channel, msg, \*\*kwargs*)

## 4.27 kombu.transport.qpid

Qpid transport using `qpid-python` as the client and `qpid-tools` for broker management.

The use this transport you must install the necessary dependencies. These dependencies are available via PyPI and can be installed using the `pip` command:

```
$ pip install kombu[qpid]
```

or to install the requirements manually:

```
$ pip install qpid-tools qpid-python
```

---

### Python 3 and PyPy Limitations

The Qpid transport does not support Python 3 or PyPy environments due to underlying dependencies not being compatible. This version is tested and works with with Python 2.7.

---

## 4.28 Authentication

This transport supports SASL authentication with the Qpid broker. Normally, SASL mechanisms are negotiated from a client list and a server list of possible mechanisms, but in practice, different SASL client libraries give different behaviors. These different behaviors cause the expected SASL mechanism to not be selected in many cases. As such, this transport restricts the mechanism types based on Kombu's configuration according to the following table.

Broker String	SASL Mechanism
<code>qpid://hostname/</code>	ANONYMOUS
<code>qpid://username:password@hostname/</code>	PLAIN
see instructions below	EXTERNAL

The user can override the above SASL selection behaviors and specify the SASL string using the `login_method` argument to the `Connection` object. The string can be a single SASL mechanism or a space separated list of SASL mechanisms. If you are using Celery with Kombu, this can be accomplished by setting the `BROKER_LOGIN_METHOD` Celery option.

**Note:** While using SSL, Qpid users may want to override the SASL mechanism to use `EXTERNAL`. In that case, Qpid requires a username to be presented that matches the `CN` of the SSL client certificate. Ensure that the broker string contains the corresponding username. For example, if the client certificate has `CN=asdf` and the client connects to `example.com` on port 5671, the broker string should be:

```
qpid://asdf@example.com:5671/
```

---

## 4.29 Transport Options

The `transport_options` argument to the `Connection` object are passed directly to the `qpid.messaging.endpoints.Connection` as keyword arguments. These options override and replace any other default or specified values. If using Celery, this can be accomplished by setting the `BROKER_TRANSPORT_OPTIONS` Celery option.

- *Transport*
- *Connection*
- *Channel*
- *Message*

## 4.29.1 Transport

**class** kombu.transport.qpid.**Transport** (\*args, \*\*kwargs)

Kombu native transport for a Qpid broker.

Provide a native transport for Kombu that allows consumers and producers to read and write messages to/from a broker. This Transport is capable of supporting both synchronous and asynchronous reading. All writes are synchronous through the *Channel* objects that support this Transport.

Asynchronous reads are done using a call to *drain\_events()*, which synchronously reads messages that were fetched asynchronously, and then handles them through calls to the callback handlers maintained on the *Connection* object.

The Transport also provides methods to establish and close a connection to the broker. This Transport establishes a factory-like pattern that allows for singleton pattern to consolidate all Connections into a single one.

The Transport can create *Channel* objects to communicate with the broker with using the *create\_channel()* method.

The Transport identifies recoverable connection errors and recoverable channel errors according to the Kombu 3.0 interface. These exception are listed as tuples and store in the Transport class attribute *recoverable\_connection\_errors* and *recoverable\_channel\_errors* respectively. Any exception raised that is not a member of one of these tuples is considered non-recoverable. This allows Kombu support for automatic retry of certain operations to function correctly.

For backwards compatibility to the pre Kombu 3.0 exception interface, the recoverable errors are also listed as *connection\_errors* and *channel\_errors*.

**class** **Connection** (\*\*connection\_options)

Encapsulate a connection object for the *Transport*.

### Parameters

- **host** – The host that connections should connect to.
- **port** – The port that connection should connect to.
- **username** – The username that connections should connect with. Optional.
- **password** – The password that connections should connect with. Optional but requires a username.
- **transport** – The transport type that connections should use. Either ‘tcp’, or ‘ssl’ are expected as values.
- **timeout** – the timeout used when a Connection connects to the broker.
- **sasl\_mechanisms** – The sasl authentication mechanism type to use. refer to SASL documentation for an explanation of valid values.

---

**Note:** qpid.messaging has an AuthenticationFailure exception type, but instead raises a ConnectionError with a message that indicates an authentication failure occurred in those situations. ConnectionError is listed as a recoverable error type, so kombu will attempt to retry if a ConnectionError is raised. Retrying

the operation without adjusting the credentials is not correct, so this method specifically checks for a `ConnectionError` that indicates an Authentication Failure occurred. In those situations, the error type is mutated while preserving the original message and raised so kombu will allow the exception to not be considered recoverable.

---

A connection object is created by a `Transport` during a call to `establish_connection()`. The `Transport` passes in connection options as keywords that should be used for any connections created. Each `Transport` creates exactly one `Connection`.

A `Connection` object maintains a reference to a `Connection` which can be accessed through a bound getter method named `get_qpuid_connection()` method. Each `Channel` uses a the `Connection` for each `BrokerAgent`, and the `Transport` maintains a session for all senders and receivers.

The `Connection` object is also responsible for maintaining the dictionary of references to callbacks that should be called when messages are received. These callbacks are saved in `_callbacks`, and keyed on the queue name associated with the received message. The `_callbacks` are setup in `Channel.basic_consume()`, removed in `Channel.basic_cancel()`, and called in `Transport.drain_events()`.

The following keys are expected to be passed in as keyword arguments at a minimum:

All keyword arguments are collected into the `connection_options` dict and passed directly through to `qpuid.messaging.endpoints.Connection.establish()`.

**class Channel** (*connection, transport*)

Supports broker configuration and messaging send and receive.

**Parameters**

- **connection** (`kombu.transport.qpuid.Connection`) – A `Connection` object that this `Channel` can reference. Currently only used to access callbacks.
- **transport** (`kombu.transport.qpuid.Transport`) – The `Transport` this `Channel` is associated with.

A channel object is designed to have method-parity with a `Channel` as defined in AMQP 0-10 and earlier, which allows for the following broker actions:

- exchange declare and delete
- queue declare and delete
- queue bind and unbind operations
- queue length and purge operations
- sending/receiving/rejecting messages
- structuring, encoding, and decoding messages
- supports synchronous and asynchronous reads
- reading state about the exchange, queues, and bindings

Channels are designed to all share a single TCP connection with a broker, but provide a level of isolated communication with the broker while benefiting from a shared TCP connection. The `Channel` is given its `Connection` object by the `Transport` that instantiates the channel.

This channel inherits from `StdChannel`, which makes this a ‘native’ channel versus a ‘virtual’ channel which would inherit from `kombu.transports.virtual`.

Messages sent using this channel are assigned a `delivery_tag`. The `delivery_tag` is generated for a message as they are prepared for sending by `basic_publish()`. The `delivery_tag` is unique per channel instance. The `delivery_tag` has no meaningful context in other objects, and is only maintained in the memory of this object, and the underlying `QoS` object that provides support.

Each channel object instantiates exactly one `QoS` object for prefetch limiting, and asynchronous ACK-ing. The `QoS` object is lazily instantiated through a property method `qos()`. The `QoS` object is a supporting object that should not be accessed directly except by the channel itself.

Synchronous reads on a queue are done using a call to `basic_get()` which uses `_get()` to

perform the reading. These methods read immediately and do not accept any form of timeout. `basic_get()` reads synchronously and ACKs messages before returning them. ACKing is done in all cases, because an application that reads messages using `qpid.messaging`, but does not ACK them will experience a memory leak. The `no_ack` argument to `basic_get()` does not affect ACKing functionality.

Asynchronous reads on a queue are done by starting a consumer using `basic_consume()`. Each call to `basic_consume()` will cause a `Receiver` to be created on the `Session` started by the `:class: Transport`. The receiver will asynchronously read using `qpid.messaging`, and prefetch messages before the call to `Transport.basic_drain()` occurs. The `prefetch_count` value of the `QoS` object is the capacity value of the new receiver. The new receiver capacity must always be at least 1, otherwise none of the receivers will appear to be ready for reading, and will never be read from.

Each call to `basic_consume()` creates a consumer, which is given a consumer tag that is identified by the caller of `basic_consume()`. Already started consumers can be cancelled using by their `consumer_tag` using `basic_cancel()`. Cancellation of a consumer causes the `Receiver` object to be closed.

Asynchronous message ACKing is supported through `basic_ack()`, and is referenced by `delivery_tag`. The `Channel` object uses its `QoS` object to perform the message ACKing.

**class** `Message` (*channel, payload, \*\*kwargs*)

**serializable** ()

**class** `Transport.Connection.Channel.QoS` (*session, prefetch\_count=1*)

A helper object for message prefetch and ACKing purposes.

**Parameters** `prefetch_count` (*int*) – Initial prefetch count, hard set to 1.

NOTE: `prefetch_count` is currently hard set to 1, and needs to be improved

This object is instantiated 1-for-1 with a `Channel` instance. `QoS` allows `prefetch_count` to be set to the number of outstanding messages the corresponding `Channel` should be allowed to prefetch. Setting `prefetch_count` to 0 disables prefetch limits, and the object can hold an arbitrary number of messages.

Messages are added using `append()`, which are held until they are ACKed asynchronously through a call to `ack()`. Messages that are received, but not ACKed will not be delivered by the broker to another consumer until an ACK is received, or the session is closed. Messages are referred to using `delivery_tag`, which are unique per `Channel`. Delivery tags are managed outside of this object and are passed in with a message to `append()`. Un-ACKed messages can be looked up from `QoS` using `get()` and can be rejected and forgotten using `reject()`.

**ack** (*delivery\_tag*)

Acknowledge a message by `delivery_tag`.

Called asynchronously once the message has been handled and can be forgotten by the broker.

**Parameters** `delivery_tag` (*uuid.UUID*) – the delivery tag associated with the message to be acknowledged.

**append** (*message, delivery\_tag*)

Append message to the list of un-ACKed messages.

Add a message, referenced by the `delivery_tag`, for ACKing, rejecting, or getting later. Messages are saved into an `collections.OrderedDict` by `delivery_tag`.

**Parameters**

- **message** (*qpid.messaging.Message*) – A received message that has not yet been ACKed.

- **delivery\_tag** (*uuid.UUID*) – A UUID to refer to this message by upon receipt.

**can\_consume** ()

Return True if the *Channel* can consume more messages, else False.

Used to ensure the client adheres to currently active prefetch limits.

**Returns** True, if this QoS object can accept more messages without violating the *prefetch\_count*. If *prefetch\_count* is 0, *can\_consume* will always return True.

**Return type** bool

**can\_consume\_max\_estimate** ()

Return the remaining message capacity for the associated *kombu.transport.qpid.Channel*.

Returns an estimated number of outstanding messages that a *kombu.transport.qpid.Channel* can accept without exceeding *prefetch\_count*. If *prefetch\_count* is 0, then this method returns 1.

**Returns** The number of estimated messages that can be fetched without violating the *prefetch\_count*.

**Return type** int

**get** (*delivery\_tag*)

Get an un-ACKed message by *delivery\_tag*. If called with an invalid *delivery\_tag* a *KeyError* is raised.

**Parameters** **delivery\_tag** (*uuid.UUID*) – The delivery tag associated with the message to be returned.

**Returns** An un-ACKed message that is looked up by *delivery\_tag*.

**Return type** *qpid.messaging.Message*

**reject** (*delivery\_tag*, *requeue=False*)

Reject a message by *delivery\_tag*.

Explicitly notify the broker that the channel associated with this QoS object is rejecting the message that was previously delivered.

If *requeue* is False, then the message is not requeued for delivery to another consumer. If *requeue* is True, then the message is requeued for delivery to another consumer.

**Parameters**

- **delivery\_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.
- **requeue** (*bool*) – If True, the broker will be notified to requeue the message. If False, the broker will be told to drop the message entirely. In both cases, the message will be removed from this object.

*Transport.Connection.Channel.basic\_ack* (*delivery\_tag*)

Acknowledge a message by *delivery\_tag*.

Acknowledges a message referenced by *delivery\_tag*. Messages can only be ACKed using *basic\_ack()* if they were acquired using *basic\_consume()*. This is the ACKing portion of the asynchronous read behavior.

Internally, this method uses the *QoS* object, which stores messages and is responsible for the ACKing.

**Parameters** **delivery\_tag** (*uuid.UUID*) – The delivery tag associated with the message to be acknowledged.

*Transport.Connection.Channel.basic\_cancel* (*consumer\_tag*)

Cancel consumer by *consumer\_tag*.



Request the consumer stops reading messages from its queue. The consumer is a `Receiver`, and it is closed using `close()`.

This method also cleans up all lingering references of the consumer.

**Parameters** `consumer_tag` (an *immutable object*) – The tag which refers to the consumer to be cancelled. Originally specified when the consumer was created as a parameter to `basic_consume()`.

`Transport.Connection.Channel.basic_consume` (*queue, no\_ack, callback, consumer\_tag, \*\*kwargs*)

Start an asynchronous consumer that reads from a queue.

This method starts a consumer of type `Receiver` using the `Session` created and referenced by the `Transport` that reads messages from a queue specified by name until stopped by a call to `basic_cancel()`.

Messages are available later through a synchronous call to `Transport.drain_events()`, which will drain from the consumer started by this method. `Transport.drain_events()` is synchronous, but the receiving of messages over the network occurs asynchronously, so it should still perform well. `Transport.drain_events()` calls the callback provided here with the `Message` of type `self.Message`.

Each consumer is referenced by a `consumer_tag`, which is provided by the caller of this method.

This method sets up the callback onto the `self.connection` object in a dict keyed by queue name. `drain_events()` is responsible for calling that callback upon message receipt.

All messages that are received are added to the `QoS` object to be saved for asynchronous ACKing later after the message has been handled by the caller of `drain_events()`. Messages can be ACKed after being received through a call to `basic_ack()`.

If `no_ack` is `True`, The `no_ack` flag indicates that the receiver of the message will not call `basic_ack()` later. Since the message will not be ACKed later, it is ACKed immediately.

`basic_consume()` transforms the message object type prior to calling the callback. Initially the message comes in as a `qpido.messaging.Message`. This method unpacks the payload of the `qpido.messaging.Message` and creates a new object of type `self.Message`.

This method wraps the user delivered callback in a runtime-built function which provides the type transformation from `qpido.messaging.Message` to `Message`, and adds the message to the associated `QoS` object for asynchronous ACKing if necessary.

#### Parameters

- **queue** (*str*) – The name of the queue to consume messages from
- **no\_ack** (*bool*) – If `True`, then messages will not be saved for ACKing later, but will be ACKed immediately. If `False`, then messages will be saved for ACKing later with a call to `basic_ack()`.
- **callback** (*a callable object*) – a callable that will be called when messages arrive on the queue.
- **consumer\_tag** (*an immutable object*) – a tag to reference the created consumer by. This `consumer_tag` is needed to cancel the consumer.

`Transport.Connection.Channel.basic_get` (*queue, no\_ack=False, \*\*kwargs*)

Non-blocking single message get and ACK from a queue by name.

Internally this method uses `_get()` to fetch the message. If an `Empty` exception is raised by `_get()`, this method silences it and returns `None`. If `_get()` does return a message, that message is ACKed. The `no_ack` parameter has no effect on ACKing behavior, and all messages

are ACKed in all cases. This method never adds fetched Messages to the internal QoS object for asynchronous ACKing.

This method converts the object type of the method as it passes through. Fetching from the broker, `_get()` returns a `qpuid.messaging.Message`, but this method takes the payload of the `qpuid.messaging.Message` and instantiates a `Message` object with the payload based on the class setting of `self.Message`.

#### Parameters

- **queue** (*str*) – The queue name to fetch a message from.
- **no\_ack** – The `no_ack` parameter has no effect on the ACK behavior of this method. Un-ACKed messages create a memory leak in `qpuid.messaging`, and need to be ACKed in all cases.

**Returns** The received message.

**Return type** `Message`

`Transport.Connection.Channel.basic_publish`(*message*, *exchange*, *routing\_key*, *\*\*kwargs*)

Publish message onto an exchange using a routing key.

Publish a message onto an exchange specified by name using a routing key specified by `routing_key`. Prepares the message in the following ways before sending:

- encodes the body using `encode_body()`
- **wraps the body as a buffer object, so that** `qpuid.messaging.endpoints.Sender` uses a content type that can support arbitrarily large messages.
- sets `delivery_tag` to a random `uuid.UUID`
- sets the exchange and routing\_key info as `delivery_info`

Internally uses `_put()` to send the message synchronously. This message is typically called by `kombu.messaging.Producer._publish` as the final step in message publication.

#### Parameters

- **message** (*dict*) – A dict containing key value pairs with the message data. A valid message dict can be generated using the `prepare_message()` method.
- **exchange** (*str*) – The name of the exchange to submit this message onto.
- **routing\_key** (*str*) – The routing key to be used as the message is submitted onto the exchange.

`Transport.Connection.Channel.basic_qos`(*prefetch\_count*, *\*args*)

Change *QoS* settings for this Channel.

Set the number of un-acknowledged messages this Channel can fetch and hold. The `prefetch_value` is also used as the capacity for any new `Receiver` objects.

Currently, this value is hard coded to 1.

**Parameters** `prefetch_count` (*int*) – Not used. This method is hard-coded to 1.

`Transport.Connection.Channel.basic_reject`(*delivery\_tag*, *requeue=False*)

Reject a message by `delivery_tag`.

Rejects a message that has been received by the Channel, but not yet acknowledged. Messages are referenced by their `delivery_tag`.

If `requeue` is `False`, the rejected message will be dropped by the broker and not delivered to any other consumers. If `requeue` is `True`, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

#### Parameters

- **delivery\_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.
- **requeue** (*bool*) – If `False`, the rejected message will be dropped by the broker and not delivered to any other consumers. If `True`, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

`Transport.Connection.Channel.body_encoding = 'base64'`

`Transport.Connection.Channel.close()`

Cancel all associated messages and close the Channel.

This cancels all consumers by calling `basic_cancel()` for each known `consumer_tag`. It also closes the `self._broker` sessions. Closing the sessions implicitly causes all outstanding, un-ACKed messages to be considered undelivered by the broker.

`Transport.Connection.Channel.codecs = {'base64': <kombu.transport.virtual.Base64 object at 0x7fb...`

`Transport.Connection.Channel.decode_body(body, encoding=None)`

Decode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in `self.codecs`. `self.codecs` uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through `encode` and `decode` methods.

#### Parameters

- **body** (*str*) – The body to be encoded.
- **encoding** (*str*) – The encoding type to be used. Must be a supported codec listed in `self.codecs`.

**Returns** If encoding is specified, the decoded body is returned. If encoding is not specified, the body is returned unchanged.

**Return type** *str*

`Transport.Connection.Channel.encode_body(body, encoding=None)`

Encode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in `self.codecs`. `self.codecs` uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through `encode` and `decode` methods.

#### Parameters

- **body** (*str*) – The body to be encoded.
- **encoding** (*str*) – The encoding type to be used. Must be a supported codec listed in `self.codecs`.

**Returns** If encoding is specified, return a tuple with the first position being the encoded body, and the second position the encoding used. If encoding is not specified, the body is passed through unchanged.

**Return type** *tuple*

```
Transport.Connection.Channel.exchange_declare(exchange='',  
                                                type='direct',  
                                                durable=False,  
                                                **kwargs)
```

Create a new exchange.

Create an exchange of a specific type, and optionally have the exchange be durable. If an exchange of the requested name already exists, no action is taken and no exceptions are raised. Durable exchanges will survive a broker restart, non-durable exchanges will not.

Exchanges provide behaviors based on their type. The expected behaviors are those defined in the AMQP 0-10 and prior specifications including 'direct', 'topic', and 'fanout' functionality.

#### Parameters

- **type** (*str*) – The exchange type. Valid values include 'direct', 'topic', and 'fanout'.
- **exchange** (*str*) – The name of the exchange to be created. If no exchange is specified, then a blank string will be used as the name.
- **durable** (*bool*) – True if the exchange should be durable, or False otherwise.

```
Transport.Connection.Channel.exchange_delete(exchange_name,  
                                              **kwargs)
```

Delete an exchange specified by name

**Parameters** **exchange\_name** (*str*) – The name of the exchange to be deleted.

```
Transport.Connection.Channel.prepare_message(body,          priority=None,  
                                              content_type=None, content_encoding=None,  
                                              headers=None,   properties=None)
```

Prepare message data for sending.

This message is typically called by `kombu.messaging.Producer._publish()` as a preparation step in message publication.

#### Parameters

- **body** (*str*) – The body of the message
- **priority** (*int*) – A number between 0 and 9 that sets the priority of the message.
- **content\_type** (*str*) – The `content_type` the message body should be treated as. If this is unset, the `qpid.messaging.endpoints.Sender` object tries to autodetect the `content_type` from the body.
- **content\_encoding** (*str*) – The `content_encoding` the message body is encoded as.
- **headers** (*dict*) – Additional Message headers that should be set. Passed in as a key-value pair.
- **properties** (*dict*) – Message properties to be set on the message.

**Returns** Returns a dict object that encapsulates message attributes. See parameters for more details on attributes that can be set.

**Return type** `dict`

`Transport.Connection.Channel.qos`  
*QoS* manager for this channel.

Lazily instantiates an object of type *QoS* upon access to the `self.qos` attribute.

**Returns** An already existing, or newly created QoS object

**Return type** *QoS*

`Transport.Connection.Channel.queue_bind`(*queue*, *exchange*, *routing\_key*,  
*\*\*kwargs*)

Bind a queue to an exchange with a bind key.

Bind a queue specified by name, to an exchange specified by name, with a specific bind key. The queue and exchange must already exist on the broker for the bind to complete successfully. Queues may be bound to exchanges multiple times with different keys.

#### Parameters

- **queue** (*str*) – The name of the queue to be bound.
- **exchange** (*str*) – The name of the exchange that the queue should be bound to.
- **routing\_key** (*str*) – The bind key that the specified queue should bind to the specified exchange with.

`Transport.Connection.Channel.queue_declare`(*queue*, *passive=False*,  
*durable=False*, *exclusive=False*, *auto\_delete=True*,  
*nowait=False*, *arguments=None*)

Create a new queue specified by name.

If the queue already exists, no change is made to the queue, and the return value returns information about the existing queue.

The queue name is required and specified as the first argument.

If `passive` is `True`, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state. Default is `False`.

If `durable` is `True`, the queue will be durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue. Default is `False`.

If `exclusive` is `True`, the queue will be exclusive. Exclusive queues may only be consumed by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'. Default is `False`.

If `auto_delete` is `True`, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted. Default is `True`.

The `nowait` parameter is unused. It was part of the 0-9-1 protocol, but this AMQP client implements 0-10 which removed the `nowait` option.

The `arguments` parameter is a set of arguments for the declaration of the queue. Arguments are passed as a dict or `None`. This field is ignored if `passive` is `True`. Default is `None`.

This method returns a `namedtuple` with the name 'queue\_declare\_ok\_t' and the queue name as 'queue', message count on the queue as 'message\_count', and the number of active consumers as 'consumer\_count'. The named tuple values are ordered as queue, message\_count, and consumer\_count respectively.

Due to Celery's non-ACKing of events, a ring policy is set on any queue that starts with the string 'celeryev' or ends with the string 'pidbox'. These are celery event queues, and Celery does not ack them, causing the messages to build-up. Eventually Qpid stops serving messages unless the 'ring' policy is set, at which point the buffer backing the queue becomes circular.

#### Parameters

- **queue** (*str*) – The name of the queue to be created.
- **passive** (*bool*) – If True, the sever will not create the queue.
- **durable** (*bool*) – If True, the queue will be durable.
- **exclusive** (*bool*) – If True, the queue will be exclusive.
- **auto\_delete** (*bool*) – If True, the queue is deleted when all consumers have finished using it.
- **nowait** (*bool*) – This parameter is unused since the 0-10 specification does not include it.
- **arguments** (*dict or None*) – A set of arguments for the declaration of the queue.

**Returns** A named tuple representing the declared queue as a named tuple. The tuple values are ordered as queue, message count, and the active consumer count.

**Return type** namedtuple

```
Transport.Connection.Channel.queue_delete(queue, if_unused=False, if_empty=False, **kwargs)
```

Delete a queue by name.

Delete a queue specified by name. Using the `if_unused` keyword argument, the delete can only occur if there are 0 consumers bound to it. Using the `if_empty` keyword argument, the delete can only occur if there are 0 messages in the queue.

#### Parameters

- **queue** (*str*) – The name of the queue to be deleted.
- **if\_unused** (*bool*) – If True, delete only if the queue has 0 consumers. If False, delete a queue even with consumers bound to it.
- **if\_empty** (*bool*) – If True, only delete the queue if it is empty. If False, delete the queue if it is empty or not.

```
Transport.Connection.Channel.queue_purge(queue, **kwargs)
```

Remove all undelivered messages from queue.

Purge all undelivered messages from a queue specified by name. If the queue does not exist an exception is raised. The queue message depth is first checked, and then the broker is asked to purge that number of messages. The integer number of messages requested to be purged is returned. The actual number of messages purged may be different than the requested number of messages to purge.

Sometimes delivered messages are asked to be purged, but are not. This case fails silently, which is the correct behavior when a message that has been delivered to a different consumer, who has not ACKed the message, and still has an active session with the broker. Messages in that case are not safe for purging and will be retained by the broker. The client is unable to change this delivery behavior.

Internally, this method relies on `_purge()`.

**Parameters** `queue` (*str*) – The name of the queue which should have all messages removed.

**Returns** The number of messages requested to be purged.

**Return type** `int`

**Raises** `qpid.messaging.exceptions.NotFound` if the queue being purged cannot be found.

`Transport.Connection.Channel.queue_unbind(queue, exchange, routing_key, **kwargs)`

Unbind a queue from an exchange with a given bind key.

Unbind a queue specified by name, from an exchange specified by name, that is already bound with a bind key. The queue and exchange must already exist on the broker, and bound with the bind key for the operation to complete successfully. Queues may be bound to exchanges multiple times with different keys, thus the bind key is a required field to unbind in an explicit way.

**Parameters**

- **queue** (*str*) – The name of the queue to be unbound.
- **exchange** (*str*) – The name of the exchange that the queue should be unbound from.
- **routing\_key** (*str*) – The existing bind key between the specified queue and a specified exchange that should be unbound.

`Transport.Connection.Channel.typeof(exchange, default='direct')`

Get the exchange type.

Lookup and return the exchange type for an exchange specified by name. Exchange types are expected to be 'direct', 'topic', and 'fanout', which correspond with exchange functionality as specified in AMQP 0-10 and earlier. If the exchange cannot be found, the default exchange type is returned.

**Parameters**

- **exchange** (*str*) – The exchange to have its type lookup up.
- **default** (*str*) – The type of exchange to assume if the exchange does not exist.

**Returns** The exchange type either 'direct', 'topic', or 'fanout'.

**Return type** `str`

`Transport.Connection.close()`

Close the connection

Closing the connection will close all associated session, senders, or receivers used by the Connection.

`Transport.Connection.close_channel(channel)`

Close a Channel.

Close a channel specified by a reference to the `Channel` object.

**Parameters** `channel` (`Channel`) – Channel that should be closed.

`Transport.Connection.get_qpid_connection()`

Return the existing connection (singleton).

**Returns** The existing `qpid.messaging.Connection`

**Return type** `qpid.messaging.endpoints.Connection`

`Transport.channel_errors = (None,)`

`Transport.close_connection(connection)`

Close the `Connection` object.

**Parameters** `connection` (`kombu.transport.qpid.Connection`) – The Connection that should be closed.

`Transport.connection_errors = (None, <class 'select.error'>)`

`Transport.create_channel(connection)`

Create and return a `Channel`.

Creates a new channel, and appends the channel to the list of channels known by the Connection. Once the new channel is created, it is returned.

**Parameters** `connection` (`kombu.transport.qpid.Connection`) – The connection that should support the new `Channel`.

**Returns** The new Channel that is made.

**Return type** `kombu.transport.qpid.Channel`.

`Transport.default_connection_params`

Return a dict with default connection parameters.

These connection parameters will be used whenever the creator of Transport does not specify a required parameter.

**Returns** A dict containing the default parameters.

**Return type** `dict`

`Transport.drain_events(connection, timeout=0, **kwargs)`

Handle and call callbacks for all ready Transport messages.

Drains all events that are ready from all `Receiver` that are asynchronously fetching messages.

For each drained message, the message is called to the appropriate callback. Callbacks are organized by queue name.

**Parameters**

- **connection** (`kombu.transport.qpid.Connection`) – The `Connection` that contains the callbacks, indexed by queue name, which will be called by this method.
- **timeout** (`int`) – The timeout that limits how long this method will run for. The timeout could interrupt a blocking read that is waiting for a new message, or cause this method to return before all messages are drained. Defaults to 0.

`Transport.driver_name = 'qpid'`

`Transport.driver_type = 'qpid'`

`Transport.establish_connection()`

Establish a Connection object.

Determines the correct options to use when creating any connections needed by this Transport, and create a `Connection` object which saves those values for connections generated as they are needed. The options are a mixture of what is passed in through the creator of the Transport, and the defaults provided by `default_connection_params()`. Options cover broker network settings, timeout behaviors, authentication, and identity verification settings.



This method also creates and stores a `Session` using the `Connection` created by this method. The `Session` is stored on `self`.

**Returns** The created `Connection` object is returned.

**Return type** `Connection`

`Transport.on_readable(connection, loop)`

Handle any messages associated with this `Transport`.

This method clears a single message from the externally monitored file descriptor by issuing a read call to the `self.r` file descriptor which removes a single '0' character that was placed into the pipe by the Qpid session message callback handler. Once a '0' is read, all available events are drained through a call to `drain_events()`.

The file descriptor `self.r` is modified to be non-blocking, ensuring that an accidental call to this method when no more messages will not cause indefinite blocking.

Nothing is expected to be returned from `drain_events()` because `drain_events()` handles messages by calling callbacks that are maintained on the `Connection` object. When `drain_events()` returns, all associated messages have been handled.

This method calls `drain_events()` which reads as many messages as are available for this `Transport`, and then returns. It blocks in the sense that reading and handling a large number of messages may take time, but it does not block waiting for a new message to arrive. When `drain_events()` is called a timeout is not specified, which causes this behavior.

One interesting behavior of note is where multiple messages are ready, and this method removes a single '0' character from `self.r`, but `drain_events()` may handle an arbitrary amount of messages. In that case, extra '0' characters may be left on `self.r` to be read, where messages corresponding with those '0' characters have already been handled. The external epoll loop will incorrectly think additional data is ready for reading, and will call `on_readable` unnecessarily, once for each '0' to be read. Additional calls to `on_readable()` produce no negative side effects, and will eventually clear out the symbols from the `self.r` file descriptor. If new messages show up during this draining period, they will also be properly handled.

#### Parameters

- **connection** (`kombu.transport.qpid.Connection`) – The connection associated with the readable events, which contains the callbacks that need to be called for the readable objects.
- **loop** (`kombu.async.Hub`) – The asynchronous loop object that contains epoll like functionality.

`Transport.polling_interval = None`

`Transport.recoverable_channel_errors = (None,)`

`Transport.recoverable_connection_errors = (None, <class 'select.error'>)`

`Transport.register_with_event_loop(connection, loop)`

Register a file descriptor and callback with the loop.

Register the callback `self.on_readable` to be called when an external epoll loop sees that the file descriptor registered is ready for reading. The file descriptor is created by this `Transport`, and is written to when a message is available.

Because `supports_ev == True`, Celery expects to call this method to give the `Transport` an opportunity to register a read file descriptor for external monitoring by celery using an Event I/O notification mechanism such as epoll. A callback is also registered that is to be called once the external epoll loop is ready to handle the epoll event associated with messages that are ready to be handled for this `Transport`.

The registration call is made exactly once per Transport after the Transport is instantiated.

#### Parameters

- **connection** (`kombu.transport.qpid.Connection`) – A reference to the connection associated with this Transport.
- **loop** (`kombu.async.hub.Hub`) – A reference to the external loop.

`Transport.supports_ev = True`

`Transport.verify_runtime_environment()`

Verify that the runtime environment is acceptable.

This method is called as part of `__init__` and raises a `RuntimeError` in Python3 or PyPi environments. This module is not compatible with Python3 or PyPi. The `RuntimeError` identifies this to the user up front along with suggesting Python 2.6+ be used instead.

This method also checks that the dependencies `qpidtoollibs` and `qpid.messaging` are installed. If either one is not installed a `RuntimeError` is raised.

**Raises** `RuntimeError` if the runtime environment is not acceptable.

## 4.29.2 Connection

`class kombu.transport.qpid.Connection(**connection_options)`

Encapsulate a connection object for the *Transport*.

#### Parameters

- **host** – The host that connections should connect to.
- **port** – The port that connection should connect to.
- **username** – The username that connections should connect with. Optional.
- **password** – The password that connections should connect with. Optional but requires a username.
- **transport** – The transport type that connections should use. Either ‘tcp’, or ‘ssl’ are expected as values.
- **timeout** – the timeout used when a Connection connects to the broker.
- **sasl\_mechanisms** – The sasl authentication mechanism type to use. refer to SASL documentation for an explanation of valid values.

---

**Note:** `qpid.messaging` has an `AuthenticationFailure` exception type, but instead raises a `ConnectionError` with a message that indicates an authentication failure occurred in those situations. `ConnectionError` is listed as a recoverable error type, so kombu will attempt to retry if a `ConnectionError` is raised. Retrying the operation without adjusting the credentials is not correct, so this method specifically checks for a `ConnectionError` that indicates an `AuthenticationFailure` occurred. In those situations, the error type is mutated while preserving the original message and raised so kombu will allow the exception to not be considered recoverable.

---

A connection object is created by a *Transport* during a call to `establish_connection()`. The *Transport* passes in connection options as keywords that should be used for any connections created. Each *Transport* creates exactly one `Connection`.

A `Connection` object maintains a reference to a `Connection` which can be accessed through a bound getter method named `get_qpid_connection()` method. Each `Channel` uses a the `Connection` for each `BrokerAgent`, and the `Transport` maintains a session for all senders and receivers.

The Connection object is also responsible for maintaining the dictionary of references to callbacks that should be called when messages are received. These callbacks are saved in `_callbacks`, and keyed on the queue name associated with the received message. The `_callbacks` are setup in `Channel.basic_consume()`, removed in `Channel.basic_cancel()`, and called in `Transport.drain_events()`.

The following keys are expected to be passed in as keyword arguments at a minimum:

All keyword arguments are collected into the `connection_options` dict and passed directly through to `qpid.messaging.endpoints.Connection.establish()`.

**class Channel** (*connection, transport*)

Supports broker configuration and messaging send and receive.

#### Parameters

- **connection** (`kombu.transport.qpid.Connection`) – A Connection object that this Channel can reference. Currently only used to access callbacks.
- **transport** (`kombu.transport.qpid.Transport`) – The Transport this Channel is associated with.

A channel object is designed to have method-parity with a Channel as defined in AMQP 0-10 and earlier, which allows for the following broker actions:

- exchange declare and delete
- queue declare and delete
- queue bind and unbind operations
- queue length and purge operations
- sending/receiving/rejecting messages
- structuring, encoding, and decoding messages
- supports synchronous and asynchronous reads
- reading state about the exchange, queues, and bindings

Channels are designed to all share a single TCP connection with a broker, but provide a level of isolated communication with the broker while benefiting from a shared TCP connection. The Channel is given its `Connection` object by the `Transport` that instantiates the channel.

This channel inherits from `StdChannel`, which makes this a ‘native’ channel versus a ‘virtual’ channel which would inherit from `kombu.transports.virtual`.

Messages sent using this channel are assigned a `delivery_tag`. The `delivery_tag` is generated for a message as they are prepared for sending by `basic_publish()`. The `delivery_tag` is unique per channel instance. The `delivery_tag` has no meaningful context in other objects, and is only maintained in the memory of this object, and the underlying `QoS` object that provides support.

Each channel object instantiates exactly one `QoS` object for prefetch limiting, and asynchronous ACKing. The `QoS` object is lazily instantiated through a property method `qos()`. The `QoS` object is a supporting object that should not be accessed directly except by the channel itself.

Synchronous reads on a queue are done using a call to `basic_get()` which uses `_get()` to perform the reading. These methods read immediately and do not accept any form of timeout. `basic_get()` reads synchronously and ACKs messages before returning them. ACKing is done in all cases, because an application that reads messages using `qpid.messaging`, but does not ACK them will experience a memory leak. The `no_ack` argument to `basic_get()` does not affect ACKing functionality.

Asynchronous reads on a queue are done by starting a consumer using `basic_consume()`. Each call to `basic_consume()` will cause a Receiver to be created on the Session started by the `:class`:

*Transport*. The receiver will asynchronously read using `qpid.messaging`, and prefetch messages before the call to `Transport.basic_drain()` occurs. The `prefetch_count` value of the *QoS* object is the capacity value of the new receiver. The new receiver capacity must always be at least 1, otherwise none of the receivers will appear to be ready for reading, and will never be read from.

Each call to `basic_consume()` creates a consumer, which is given a consumer tag that is identified by the caller of `basic_consume()`. Already started consumers can be cancelled using by their `consumer_tag` using `basic_cancel()`. Cancellation of a consumer causes the `Receiver` object to be closed.

Asynchronous message ACKing is supported through `basic_ack()`, and is referenced by `delivery_tag`. The `Channel` object uses its *QoS* object to perform the message ACKing.

**class Message** (*channel, payload, \*\*kwargs*)

**serializable** ()

**class** `Connection.Channel.QoS` (*session, prefetch\_count=1*)

A helper object for message prefetch and ACKing purposes.

**Parameters** `prefetch_count` (*int*) – Initial prefetch count, hard set to 1.

NOTE: `prefetch_count` is currently hard set to 1, and needs to be improved

This object is instantiated 1-for-1 with a *Channel* instance. QoS allows `prefetch_count` to be set to the number of outstanding messages the corresponding *Channel* should be allowed to prefetch. Setting `prefetch_count` to 0 disables prefetch limits, and the object can hold an arbitrary number of messages.

Messages are added using `append()`, which are held until they are ACKed asynchronously through a call to `ack()`. Messages that are received, but not ACKed will not be delivered by the broker to another consumer until an ACK is received, or the session is closed. Messages are referred to using `delivery_tag`, which are unique per *Channel*. Delivery tags are managed outside of this object and are passed in with a message to `append()`. Un-ACKed messages can be looked up from QoS using `get()` and can be rejected and forgotten using `reject()`.

**ack** (*delivery\_tag*)

Acknowledge a message by `delivery_tag`.

Called asynchronously once the message has been handled and can be forgotten by the broker.

**Parameters** `delivery_tag` (*uuid.UUID*) – the delivery tag associated with the message to be acknowledged.

**append** (*message, delivery\_tag*)

Append message to the list of un-ACKed messages.

Add a message, referenced by the `delivery_tag`, for ACKing, rejecting, or getting later. Messages are saved into an `collections.OrderedDict` by `delivery_tag`.

**Parameters**

- **message** (*qpid.messaging.Message*) – A received message that has not yet been ACKed.
- **delivery\_tag** (*uuid.UUID*) – A UUID to refer to this message by upon receipt.

**can\_consume** ()

Return True if the *Channel* can consume more messages, else False.

Used to ensure the client adheres to currently active prefetch limits.

**Returns** True, if this QoS object can accept more messages without violating the `prefetch_count`. If `prefetch_count` is 0, `can_consume` will always return True.

**Return type** `bool`

**can\_consume\_max\_estimate** ()

Return the remaining message capacity for the associated `kombu.transport.qpid.Channel`.

Returns an estimated number of outstanding messages that a `kombu.transport.qpid.Channel` can accept without exceeding `prefetch_count`. If `prefetch_count` is 0, then this method returns 1.

**Returns** The number of estimated messages that can be fetched without violating the `prefetch_count`.

**Return type** `int`

**get** (*delivery\_tag*)

Get an un-ACKed message by `delivery_tag`. If called with an invalid `delivery_tag` a `KeyError` is raised.

**Parameters** `delivery_tag` (*uuid.UUID*) – The delivery tag associated with the message to be returned.

**Returns** An un-ACKed message that is looked up by `delivery_tag`.

**Return type** `qpid.messaging.Message`

**reject** (*delivery\_tag*, *requeue=False*)

Reject a message by `delivery_tag`.

Explicitly notify the broker that the channel associated with this QoS object is rejecting the message that was previously delivered.

If `requeue` is `False`, then the message is not requeued for delivery to another consumer. If `requeue` is `True`, then the message is requeued for delivery to another consumer.

**Parameters**

- **delivery\_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.
- **requeue** (*bool*) – If `True`, the broker will be notified to requeue the message. If `False`, the broker will be told to drop the message entirely. In both cases, the message will be removed from this object.

`Connection.Channel.basic_ack` (*delivery\_tag*)

Acknowledge a message by `delivery_tag`.

Acknowledges a message referenced by `delivery_tag`. Messages can only be ACKed using `basic_ack()` if they were acquired using `basic_consume()`. This is the ACKing portion of the asynchronous read behavior.

Internally, this method uses the `QoS` object, which stores messages and is responsible for the ACKing.

**Parameters** `delivery_tag` (*uuid.UUID*) – The delivery tag associated with the message to be acknowledged.

`Connection.Channel.basic_cancel` (*consumer\_tag*)

Cancel consumer by `consumer_tag`.

Request the consumer stops reading messages from its queue. The consumer is a `Receiver`, and it is closed using `close()`.

This method also cleans up all lingering references of the consumer.

**Parameters** `consumer_tag` (*an immutable object*) – The tag which refers to the consumer to be cancelled. Originally specified when the consumer was created as a parameter to `basic_consume()`.

`Connection.Channel.basic_consume(queue, no_ack, callback, consumer_tag, **kwargs)`

Start an asynchronous consumer that reads from a queue.

This method starts a consumer of type `Receiver` using the `Session` created and referenced by the `Transport` that reads messages from a queue specified by name until stopped by a call to `basic_cancel()`.

Messages are available later through a synchronous call to `Transport.drain_events()`, which will drain from the consumer started by this method. `Transport.drain_events()` is synchronous, but the receiving of messages over the network occurs asynchronously, so it should still perform well. `Transport.drain_events()` calls the callback provided here with the `Message` of type `self.Message`.

Each consumer is referenced by a `consumer_tag`, which is provided by the caller of this method.

This method sets up the callback onto the `self.connection` object in a dict keyed by queue name. `drain_events()` is responsible for calling that callback upon message receipt.

All messages that are received are added to the `QoS` object to be saved for asynchronous ACKing later after the message has been handled by the caller of `drain_events()`. Messages can be ACKed after being received through a call to `basic_ack()`.

If `no_ack` is `True`, The `no_ack` flag indicates that the receiver of the message will not call `basic_ack()` later. Since the message will not be ACKed later, it is ACKed immediately.

`basic_consume()` transforms the message object type prior to calling the callback. Initially the message comes in as a `qpido.messaging.Message`. This method unpacks the payload of the `qpido.messaging.Message` and creates a new object of type `self.Message`.

This method wraps the user delivered callback in a runtime-built function which provides the type transformation from `qpido.messaging.Message` to `Message`, and adds the message to the associated `QoS` object for asynchronous ACKing if necessary.

#### Parameters

- **queue** (*str*) – The name of the queue to consume messages from
- **no\_ack** (*bool*) – If `True`, then messages will not be saved for ACKing later, but will be ACKed immediately. If `False`, then messages will be saved for ACKing later with a call to `basic_ack()`.
- **callback** (*a callable object*) – a callable that will be called when messages arrive on the queue.
- **consumer\_tag** (*an immutable object*) – a tag to reference the created consumer by. This `consumer_tag` is needed to cancel the consumer.

`Connection.Channel.basic_get(queue, no_ack=False, **kwargs)`

Non-blocking single message get and ACK from a queue by name.

Internally this method uses `_get()` to fetch the message. If an `Empty` exception is raised by `_get()`, this method silences it and returns `None`. If `_get()` does return a message, that message is ACKed. The `no_ack` parameter has no effect on ACKing behavior, and all messages are ACKed

in all cases. This method never adds fetched Messages to the internal QoS object for asynchronous ACKing.

This method converts the object type of the method as it passes through. Fetching from the broker, `_get()` returns a `qpido.messaging.Message`, but this method takes the payload of the `qpido.messaging.Message` and instantiates a `Message` object with the payload based on the class setting of `self.Message`.

#### Parameters

- **queue** (*str*) – The queue name to fetch a message from.
- **no\_ack** – The `no_ack` parameter has no effect on the ACK behavior of this method. Un-ACKed messages create a memory leak in `qpido.messaging`, and need to be ACKed in all cases.

**Returns** The received message.

**Return type** `Message`

`Connection.Channel.basic_publish` (*message, exchange, routing\_key, \*\*kwargs*)

Publish message onto an exchange using a routing key.

Publish a message onto an exchange specified by name using a routing key specified by `routing_key`. Prepares the message in the following ways before sending:

- encodes the body using `encode_body()`
- **wraps the body as a buffer object, so that** `qpido.messaging.endpoints.Sender` uses a content type that can support arbitrarily large messages.
- sets `delivery_tag` to a random `uuid.UUID`
- sets the exchange and `routing_key` info as `delivery_info`

Internally uses `_put()` to send the message synchronously. This message is typically called by `kombu.messaging.Producer._publish` as the final step in message publication.

#### Parameters

- **message** (*dict*) – A dict containing key value pairs with the message data. A valid message dict can be generated using the `prepare_message()` method.
- **exchange** (*str*) – The name of the exchange to submit this message onto.
- **routing\_key** (*str*) – The routing key to be used as the message is submitted onto the exchange.

`Connection.Channel.basic_qos` (*prefetch\_count, \*args*)

Change *QoS* settings for this Channel.

Set the number of un-acknowledged messages this Channel can fetch and hold. The `prefetch_value` is also used as the capacity for any new `Receiver` objects.

Currently, this value is hard coded to 1.

**Parameters** `prefetch_count` (*int*) – Not used. This method is hard-coded to 1.

`Connection.Channel.basic_reject` (*delivery\_tag, requeue=False*)

Reject a message by `delivery_tag`.

Rejects a message that has been received by the Channel, but not yet acknowledged. Messages are referenced by their `delivery_tag`.

If `requeue` is `False`, the rejected message will be dropped by the broker and not delivered to any other consumers. If `requeue` is `True`, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

#### Parameters

- **delivery\_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.
- **requeue** (*bool*) – If `False`, the rejected message will be dropped by the broker and not delivered to any other consumers. If `True`, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

`Connection.Channel.body_encoding = 'base64'`

`Connection.Channel.close()`

Cancel all associated messages and close the Channel.

This cancels all consumers by calling `basic_cancel()` for each known `consumer_tag`. It also closes the `self._broker` sessions. Closing the sessions implicitly causes all outstanding, un-ACKED messages to be considered undelivered by the broker.

`Connection.Channel.codecs = {'base64': <kombu.transport.virtual.Base64 object at 0x7fb7488f4fd0>}`

`Connection.Channel.decode_body(body, encoding=None)`

Decode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in `self.codecs`. `self.codecs` uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through `encode` and `decode` methods.

#### Parameters

- **body** (*str*) – The body to be encoded.
- **encoding** (*str*) – The encoding type to be used. Must be a supported codec listed in `self.codecs`.

**Returns** If encoding is specified, the decoded body is returned. If encoding is not specified, the body is returned unchanged.

**Return type** *str*

`Connection.Channel.encode_body(body, encoding=None)`

Encode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in `self.codecs`. `self.codecs` uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through `encode` and `decode` methods.

#### Parameters

- **body** (*str*) – The body to be encoded.
- **encoding** (*str*) – The encoding type to be used. Must be a supported codec listed in `self.codecs`.

**Returns** If encoding is specified, return a tuple with the first position being the encoded body, and the second position the encoding used. If encoding is not specified, the body is passed through unchanged.

**Return type** *tuple*



`Connection.Channel.exchange_declare` (*exchange=''*, *type='direct'*, *durable=False*, *\*\*kwargs*)

Create a new exchange.

Create an exchange of a specific type, and optionally have the exchange be durable. If an exchange of the requested name already exists, no action is taken and no exceptions are raised. Durable exchanges will survive a broker restart, non-durable exchanges will not.

Exchanges provide behaviors based on their type. The expected behaviors are those defined in the AMQP 0-10 and prior specifications including 'direct', 'topic', and 'fanout' functionality.

#### Parameters

- **type** (*str*) – The exchange type. Valid values include 'direct', 'topic', and 'fanout'.
- **exchange** (*str*) – The name of the exchange to be created. If no exchange is specified, then a blank string will be used as the name.
- **durable** (*bool*) – True if the exchange should be durable, or False otherwise.

`Connection.Channel.exchange_delete` (*exchange\_name*, *\*\*kwargs*)

Delete an exchange specified by name

**Parameters** **exchange\_name** (*str*) – The name of the exchange to be deleted.

`Connection.Channel.prepare_message` (*body*, *priority=None*, *content\_type=None*, *content\_encoding=None*, *headers=None*, *properties=None*)

Prepare message data for sending.

This message is typically called by `kombu.messaging.Producer._publish()` as a preparation step in message publication.

#### Parameters

- **body** (*str*) – The body of the message
- **priority** (*int*) – A number between 0 and 9 that sets the priority of the message.
- **content\_type** (*str*) – The content\_type the message body should be treated as. If this is unset, the `qpid.messaging.endpoints.Sender` object tries to autodetect the content\_type from the body.
- **content\_encoding** (*str*) – The content\_encoding the message body is encoded as.
- **headers** (*dict*) – Additional Message headers that should be set. Passed in as a key-value pair.
- **properties** (*dict*) – Message properties to be set on the message.

**Returns** Returns a dict object that encapsulates message attributes. See parameters for more details on attributes that can be set.

**Return type** `dict`

`Connection.Channel.qos`

QoS manager for this channel.

Lazily instantiates an object of type `QoS` upon access to the `self.qos` attribute.

**Returns** An already existing, or newly created QoS object

**Return type** `QoS`

`Connection.Channel.queue_bind` (*queue, exchange, routing\_key, \*\*kwargs*)

Bind a queue to an exchange with a bind key.

Bind a queue specified by name, to an exchange specified by name, with a specific bind key. The queue and exchange must already exist on the broker for the bind to complete successfully. Queues may be bound to exchanges multiple times with different keys.

#### Parameters

- **queue** (*str*) – The name of the queue to be bound.
- **exchange** (*str*) – The name of the exchange that the queue should be bound to.
- **routing\_key** (*str*) – The bind key that the specified queue should bind to the specified exchange with.

`Connection.Channel.queue_declare` (*queue, passive=False, durable=False, exclusive=False, auto\_delete=True, nowait=False, arguments=None*)

Create a new queue specified by name.

If the queue already exists, no change is made to the queue, and the return value returns information about the existing queue.

The queue name is required and specified as the first argument.

If `passive` is `True`, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state. Default is `False`.

If `durable` is `True`, the queue will be durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue. Default is `False`.

If `exclusive` is `True`, the queue will be exclusive. Exclusive queues may only be consumed by the current connection. Setting the ‘exclusive’ flag always implies ‘auto-delete’. Default is `False`.

If `auto_delete` is `True`, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won’t be deleted. Default is `True`.

The `nowait` parameter is unused. It was part of the 0-9-1 protocol, but this AMQP client implements 0-10 which removed the `nowait` option.

The `arguments` parameter is a set of arguments for the declaration of the queue. Arguments are passed as a dict or `None`. This field is ignored if `passive` is `True`. Default is `None`.

This method returns a `namedtuple` with the name ‘`queue_declare_ok_t`’ and the queue name as ‘`queue`’, message count on the queue as ‘`message_count`’, and the number of active consumers as ‘`consumer_count`’. The named tuple values are ordered as `queue`, `message_count`, and `consumer_count` respectively.

Due to Celery’s non-ACKing of events, a ring policy is set on any queue that starts with the string ‘`celeryev`’ or ends with the string ‘`pidbox`’. These are celery event queues, and Celery does not ack them, causing the messages to build-up. Eventually Qpid stops serving messages unless the ‘ring’ policy is set, at which point the buffer backing the queue becomes circular.

#### Parameters

- **queue** (*str*) – The name of the queue to be created.
- **passive** (*bool*) – If `True`, the sever will not create the queue.

- **durable**  (*bool*) – If True, the queue will be durable.
- **exclusive**  (*bool*) – If True, the queue will be exclusive.
- **auto\_delete**  (*bool*) – If True, the queue is deleted when all consumers have finished using it.
- **nowait**  (*bool*) – This parameter is unused since the 0-10 specification does not include it.
- **arguments**  (*dict or None*) – A set of arguments for the declaration of the queue.

**Returns** A named tuple representing the declared queue as a named tuple. The tuple values are ordered as queue, message count, and the active consumer count.

**Return type** `namedtuple`

`Connection.Channel.queue_delete` (*queue*, *if\_unused=False*, *if\_empty=False*, *\*\*kwargs*)

Delete a queue by name.

Delete a queue specified by name. Using the `if_unused` keyword argument, the delete can only occur if there are 0 consumers bound to it. Using the `if_empty` keyword argument, the delete can only occur if there are 0 messages in the queue.

#### Parameters

- **queue**  (*str*) – The name of the queue to be deleted.
- **if\_unused**  (*bool*) – If True, delete only if the queue has 0 consumers. If False, delete a queue even with consumers bound to it.
- **if\_empty**  (*bool*) – If True, only delete the queue if it is empty. If False, delete the queue if it is empty or not.

`Connection.Channel.queue_purge` (*queue*, *\*\*kwargs*)

Remove all undelivered messages from queue.

Purge all undelivered messages from a queue specified by name. If the queue does not exist an exception is raised. The queue message depth is first checked, and then the broker is asked to purge that number of messages. The integer number of messages requested to be purged is returned. The actual number of messages purged may be different than the requested number of messages to purge.

Sometimes delivered messages are asked to be purged, but are not. This case fails silently, which is the correct behavior when a message that has been delivered to a different consumer, who has not ACKed the message, and still has an active session with the broker. Messages in that case are not safe for purging and will be retained by the broker. The client is unable to change this delivery behavior.

Internally, this method relies on `_purge()`.

**Parameters** **queue** (*str*) – The name of the queue which should have all messages removed.

**Returns** The number of messages requested to be purged.

**Return type** `int`

**Raises** `qpuid.messaging.exceptions.NotFound` if the queue being purged cannot be found.

`Connection.Channel.queue_unbind(queue, exchange, routing_key, **kwargs)`

Unbind a queue from an exchange with a given bind key.

Unbind a queue specified by name, from an exchange specified by name, that is already bound with a bind key. The queue and exchange must already exist on the broker, and bound with the bind key for the operation to complete successfully. Queues may be bound to exchanges multiple times with different keys, thus the bind key is a required field to unbind in an explicit way.

#### Parameters

- **queue** (*str*) – The name of the queue to be unbound.
- **exchange** (*str*) – The name of the exchange that the queue should be unbound from.
- **routing\_key** (*str*) – The existing bind key between the specified queue and a specified exchange that should be unbound.

`Connection.Channel.typeof(exchange, default='direct')`

Get the exchange type.

Lookup and return the exchange type for an exchange specified by name. Exchange types are expected to be 'direct', 'topic', and 'fanout', which correspond with exchange functionality as specified in AMQP 0-10 and earlier. If the exchange cannot be found, the default exchange type is returned.

#### Parameters

- **exchange** (*str*) – The exchange to have its type lookup up.
- **default** (*str*) – The type of exchange to assume if the exchange does not exist.

**Returns** The exchange type either 'direct', 'topic', or 'fanout'.

**Return type** *str*

`Connection.close()`

Close the connection

Closing the connection will close all associated session, senders, or receivers used by the Connection.

`Connection.close_channel(channel)`

Close a Channel.

Close a channel specified by a reference to the *Channel* object.

**Parameters** **channel** (*Channel*.) – Channel that should be closed.

`Connection.get_qpid_connection()`

Return the existing connection (singleton).

**Returns** The existing `qpid.messaging.Connection`

**Return type** `qpid.messaging.endpoints.Connection`

### 4.29.3 Channel

`class kombu.transport.qpid.Channel(connection, transport)`

Supports broker configuration and messaging send and receive.

#### Parameters

- **connection** (`kombu.transport.qpid.Connection`) – A Connection object that this Channel can reference. Currently only used to access callbacks.

- **transport** (`kombu.transport.qpid.Transport`) – The Transport this Channel is associated with.

A channel object is designed to have method-parity with a Channel as defined in AMQP 0-10 and earlier, which allows for the following broker actions:

- exchange declare and delete
- queue declare and delete
- queue bind and unbind operations
- queue length and purge operations
- sending/receiving/rejecting messages
- structuring, encoding, and decoding messages
- supports synchronous and asynchronous reads
- reading state about the exchange, queues, and bindings

Channels are designed to all share a single TCP connection with a broker, but provide a level of isolated communication with the broker while benefiting from a shared TCP connection. The Channel is given its *Connection* object by the *Transport* that instantiates the channel.

This channel inherits from `StdChannel`, which makes this a ‘native’ channel versus a ‘virtual’ channel which would inherit from `kombu.transports.virtual`.

Messages sent using this channel are assigned a `delivery_tag`. The `delivery_tag` is generated for a message as they are prepared for sending by `basic_publish()`. The `delivery_tag` is unique per channel instance. The `delivery_tag` has no meaningful context in other objects, and is only maintained in the memory of this object, and the underlying *QoS* object that provides support.

Each channel object instantiates exactly one *QoS* object for prefetch limiting, and asynchronous ACKing. The *QoS* object is lazily instantiated through a property method `qos()`. The *QoS* object is a supporting object that should not be accessed directly except by the channel itself.

Synchronous reads on a queue are done using a call to `basic_get()` which uses `_get()` to perform the reading. These methods read immediately and do not accept any form of timeout. `basic_get()` reads synchronously and ACKs messages before returning them. ACKing is done in all cases, because an application that reads messages using `qpid.messaging`, but does not ACK them will experience a memory leak. The `no_ack` argument to `basic_get()` does not affect ACKing functionality.

Asynchronous reads on a queue are done by starting a consumer using `basic_consume()`. Each call to `basic_consume()` will cause a *Receiver* to be created on the *Session* started by the `:class: Transport`. The receiver will asynchronously read using `qpid.messaging`, and prefetch messages before the call to `Transport.basic_drain()` occurs. The `prefetch_count` value of the *QoS* object is the capacity value of the new receiver. The new receiver capacity must always be at least 1, otherwise none of the receivers will appear to be ready for reading, and will never be read from.

Each call to `basic_consume()` creates a consumer, which is given a consumer tag that is identified by the caller of `basic_consume()`. Already started consumers can be cancelled using by their `consumer_tag` using `basic_cancel()`. Cancellation of a consumer causes the *Receiver* object to be closed.

Asynchronous message ACKing is supported through `basic_ack()`, and is referenced by `delivery_tag`. The Channel object uses its *QoS* object to perform the message ACKing.

**class** `Message` (*channel, payload, \*\*kwargs*)

A class reference that identifies

**serializable** ()

**class** `Channel.QoS` (*session, prefetch\_count=1*)

A class reference that will be instantiated using the `qos` property.

**ack** (*delivery\_tag*)

Acknowledge a message by `delivery_tag`.

Called asynchronously once the message has been handled and can be forgotten by the broker.

**Parameters** `delivery_tag` (*uuid.UUID*) – the delivery tag associated with the message to be acknowledged.

**append** (*message*, *delivery\_tag*)

Append message to the list of un-ACKed messages.

Add a message, referenced by the `delivery_tag`, for ACKing, rejecting, or getting later. Messages are saved into an `collections.OrderedDict` by `delivery_tag`.

**Parameters**

- **message** (*qpido.messaging.Message*) – A received message that has not yet been ACKed.
- **delivery\_tag** (*uuid.UUID*) – A UUID to refer to this message by upon receipt.

**can\_consume** ()

Return True if the *Channel* can consume more messages, else False.

Used to ensure the client adheres to currently active prefetch limits.

**Returns** True, if this QoS object can accept more messages without violating the `prefetch_count`. If `prefetch_count` is 0, `can_consume` will always return True.

**Return type** `bool`

**can\_consume\_max\_estimate** ()

Return the remaining message capacity for the associated *kombu.transport.qpid.Channel*.

Returns an estimated number of outstanding messages that a *kombu.transport.qpid.Channel* can accept without exceeding `prefetch_count`. If `prefetch_count` is 0, then this method returns 1.

**Returns** The number of estimated messages that can be fetched without violating the `prefetch_count`.

**Return type** `int`

**get** (*delivery\_tag*)

Get an un-ACKed message by `delivery_tag`. If called with an invalid `delivery_tag` a `KeyError` is raised.

**Parameters** `delivery_tag` (*uuid.UUID*) – The delivery tag associated with the message to be returned.

**Returns** An un-ACKed message that is looked up by `delivery_tag`.

**Return type** *qpido.messaging.Message*

**reject** (*delivery\_tag*, *requeue=False*)

Reject a message by `delivery_tag`.

Explicitly notify the broker that the channel associated with this QoS object is rejecting the message that was previously delivered.

If `requeue` is False, then the message is not requeued for delivery to another consumer. If `requeue` is True, then the message is requeued for delivery to another consumer.

**Parameters**

- **delivery\_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.

- **requeue** (*bool*) – If True, the broker will be notified to requeue the message. If False, the broker will be told to drop the message entirely. In both cases, the message will be removed from this object.

Channel.**basic\_ack** (*delivery\_tag*)

Acknowledge a message by *delivery\_tag*.

Acknowledges a message referenced by *delivery\_tag*. Messages can only be ACKed using *basic\_ack()* if they were acquired using *basic\_consume()*. This is the ACKing portion of the asynchronous read behavior.

Internally, this method uses the *QoS* object, which stores messages and is responsible for the ACKing.

**Parameters** *delivery\_tag* (*uuid.UUID*) – The delivery tag associated with the message to be acknowledged.

Channel.**basic\_cancel** (*consumer\_tag*)

Cancel consumer by consumer tag.

Request the consumer stops reading messages from its queue. The consumer is a *Receiver*, and it is closed using *close()*.

This method also cleans up all lingering references of the consumer.

**Parameters** *consumer\_tag* (*an immutable object*) – The tag which refers to the consumer to be cancelled. Originally specified when the consumer was created as a parameter to *basic\_consume()*.

Channel.**basic\_consume** (*queue, no\_ack, callback, consumer\_tag, \*\*kwargs*)

Start an asynchronous consumer that reads from a queue.

This method starts a consumer of type *Receiver* using the *Session* created and referenced by the *Transport* that reads messages from a queue specified by name until stopped by a call to *basic\_cancel()*.

Messages are available later through a synchronous call to *Transport.drain\_events()*, which will drain from the consumer started by this method. *Transport.drain\_events()* is synchronous, but the receiving of messages over the network occurs asynchronously, so it should still perform well. *Transport.drain\_events()* calls the callback provided here with the *Message* of type *self.Message*.

Each consumer is referenced by a *consumer\_tag*, which is provided by the caller of this method.

This method sets up the callback onto the *self.connection* object in a dict keyed by queue name. *drain\_events()* is responsible for calling that callback upon message receipt.

All messages that are received are added to the *QoS* object to be saved for asynchronous ACKing later after the message has been handled by the caller of *drain\_events()*. Messages can be ACKed after being received through a call to *basic\_ack()*.

If *no\_ack* is True, The *no\_ack* flag indicates that the receiver of the message will not call *basic\_ack()* later. Since the message will not be ACKed later, it is ACKed immediately.

*basic\_consume()* transforms the message object type prior to calling the callback. Initially the message comes in as a *qpuid.messaging.Message*. This method unpacks the payload of the *qpuid.messaging.Message* and creates a new object of type *self.Message*.

This method wraps the user delivered callback in a runtime-built function which provides the type transformation from *qpuid.messaging.Message* to *Message*, and adds the message to the associated *QoS* object for asynchronous ACKing if necessary.

**Parameters**

- **queue** (*str*) – The name of the queue to consume messages from
- **no\_ack** (*bool*) – If True, then messages will not be saved for ACKing later, but will be ACKed immediately. If False, then messages will be saved for ACKing later with a call to `basic_ack()`.
- **callback** (*a callable object*) – a callable that will be called when messages arrive on the queue.
- **consumer\_tag** (*an immutable object*) – a tag to reference the created consumer by. This `consumer_tag` is needed to cancel the consumer.

`Channel.basic_get` (*queue, no\_ack=False, \*\*kwargs*)

Non-blocking single message get and ACK from a queue by name.

Internally this method uses `_get()` to fetch the message. If an `Empty` exception is raised by `_get()`, this method silences it and returns `None`. If `_get()` does return a message, that message is ACKed. The `no_ack` parameter has no effect on ACKing behavior, and all messages are ACKed in all cases. This method never adds fetched Messages to the internal QoS object for asynchronous ACKing.

This method converts the object type of the method as it passes through. Fetching from the broker, `_get()` returns a `qpido.messaging.Message`, but this method takes the payload of the `qpido.messaging.Message` and instantiates a `Message` object with the payload based on the class setting of `self.Message`.

#### Parameters

- **queue** (*str*) – The queue name to fetch a message from.
- **no\_ack** – The `no_ack` parameter has no effect on the ACK behavior of this method. Un-ACKed messages create a memory leak in `qpido.messaging`, and need to be ACKed in all cases.

**Returns** The received message.

**Return type** `Message`

`Channel.basic_publish` (*message, exchange, routing\_key, \*\*kwargs*)

Publish message onto an exchange using a routing key.

Publish a message onto an exchange specified by name using a routing key specified by `routing_key`. Prepares the message in the following ways before sending:

- encodes the body using `encode_body()`
- **wraps the body as a buffer object, so that** `qpido.messaging.endpoints.Sender` uses a content type that can support arbitrarily large messages.
- sets `delivery_tag` to a random `uuid.UUID`
- sets the `exchange` and `routing_key` info as `delivery_info`

Internally uses `_put()` to send the message synchronously. This message is typically called by `kombu.messaging.Producer._publish` as the final step in message publication.

#### Parameters

- **message** (*dict*) – A dict containing key value pairs with the message data. A valid message dict can be generated using the `prepare_message()` method.
- **exchange** (*str*) – The name of the exchange to submit this message onto.
- **routing\_key** (*str*) – The routing key to be used as the message is submitted onto the exchange.



`Channel.basic_qos` (*prefetch\_count*, \*args)

Change *QoS* settings for this Channel.

Set the number of un-acknowledged messages this Channel can fetch and hold. The *prefetch\_value* is also used as the capacity for any new `Receiver` objects.

Currently, this value is hard coded to 1.

**Parameters** `prefetch_count` (*int*) – Not used. This method is hard-coded to 1.

`Channel.basic_reject` (*delivery\_tag*, *requeue=False*)

Reject a message by *delivery\_tag*.

Rejects a message that has been received by the Channel, but not yet acknowledged. Messages are referenced by their *delivery\_tag*.

If *requeue* is `False`, the rejected message will be dropped by the broker and not delivered to any other consumers. If *requeue* is `True`, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

#### Parameters

- **delivery\_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.
- **requeue** (*bool*) – If `False`, the rejected message will be dropped by the broker and not delivered to any other consumers. If `True`, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

`Channel.body_encoding = 'base64'`

Default body encoding. NOTE: `transport_options['body_encoding']` will override this value.

`Channel.close()`

Cancel all associated messages and close the Channel.

This cancels all consumers by calling `basic_cancel()` for each known *consumer\_tag*. It also closes the `self._broker` sessions. Closing the sessions implicitly causes all outstanding, un-ACKed messages to be considered undelivered by the broker.

`Channel.codecs = {'base64': <kombu.transport.virtual.Base64 object at 0x7fb7488f4fd0>}`

Binary <-> ASCII codecs.

`Channel.decode_body` (*body*, *encoding=None*)

Decode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in `self.codecs`. `self.codecs` uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through `encode` and `decode` methods.

#### Parameters

- **body** (*str*) – The body to be encoded.
- **encoding** (*str*) – The encoding type to be used. Must be a supported codec listed in `self.codecs`.

**Returns** If encoding is specified, the decoded body is returned. If encoding is not specified, the body is returned unchanged.

**Return type** `str`

`Channel.encode_body` (*body*, *encoding=None*)

Encode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in `self.codecs`. `self.codecs` uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through `encode` and `decode` methods.

#### Parameters

- **body** (*str*) – The body to be encoded.
- **encoding** (*str*) – The encoding type to be used. Must be a supported codec listed in `self.codecs`.

**Returns** If encoding is specified, return a tuple with the first position being the encoded body, and the second position the encoding used. If encoding is not specified, the body is passed through unchanged.

**Return type** tuple

`Channel.exchange_declare` (*exchange=''*, *type='direct'*, *durable=False*, *\*\*kwargs*)

Create a new exchange.

Create an exchange of a specific type, and optionally have the exchange be durable. If an exchange of the requested name already exists, no action is taken and no exceptions are raised. Durable exchanges will survive a broker restart, non-durable exchanges will not.

Exchanges provide behaviors based on their type. The expected behaviors are those defined in the AMQP 0-10 and prior specifications including 'direct', 'topic', and 'fanout' functionality.

#### Parameters

- **type** (*str*) – The exchange type. Valid values include 'direct', 'topic', and 'fanout'.
- **exchange** (*str*) – The name of the exchange to be created. If no exchange is specified, then a blank string will be used as the name.
- **durable** (*bool*) – True if the exchange should be durable, or False otherwise.

`Channel.exchange_delete` (*exchange\_name*, *\*\*kwargs*)

Delete an exchange specified by name

**Parameters** **exchange\_name** (*str*) – The name of the exchange to be deleted.

`Channel.prepare_message` (*body*, *priority=None*, *content\_type=None*, *content\_encoding=None*, *headers=None*, *properties=None*)

Prepare message data for sending.

This message is typically called by `kombu.messaging.Producer._publish()` as a preparation step in message publication.

#### Parameters

- **body** (*str*) – The body of the message
- **priority** (*int*) – A number between 0 and 9 that sets the priority of the message.
- **content\_type** (*str*) – The `content_type` the message body should be treated as. If this is unset, the `qpid.messaging.endpoints.Sender` object tries to autodetect the `content_type` from the body.
- **content\_encoding** (*str*) – The `content_encoding` the message body is encoded as.

- **headers** (*dict*) – Additional Message headers that should be set. Passed in as a key-value pair.
- **properties** (*dict*) – Message properties to be set on the message.

**Returns** Returns a dict object that encapsulates message attributes. See parameters for more details on attributes that can be set.

**Return type** *dict*

`Channel.qos`

*QoS* manager for this channel.

Lazily instantiates an object of type *QoS* upon access to the `self.qos` attribute.

**Returns** An already existing, or newly created QoS object

**Return type** *QoS*

`Channel.queue_bind` (*queue, exchange, routing\_key, \*\*kwargs*)

Bind a queue to an exchange with a bind key.

Bind a queue specified by name, to an exchange specified by name, with a specific bind key. The queue and exchange must already exist on the broker for the bind to complete successfully. Queues may be bound to exchanges multiple times with different keys.

**Parameters**

- **queue** (*str*) – The name of the queue to be bound.
- **exchange** (*str*) – The name of the exchange that the queue should be bound to.
- **routing\_key** (*str*) – The bind key that the specified queue should bind to the specified exchange with.

`Channel.queue_declare` (*queue, passive=False, durable=False, exclusive=False, auto\_delete=True, nowait=False, arguments=None*)

Create a new queue specified by name.

If the queue already exists, no change is made to the queue, and the return value returns information about the existing queue.

The queue name is required and specified as the first argument.

If `passive` is `True`, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state. Default is `False`.

If `durable` is `True`, the queue will be durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue. Default is `False`.

If `exclusive` is `True`, the queue will be exclusive. Exclusive queues may only be consumed by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'. Default is `False`.

If `auto_delete` is `True`, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted. Default is `True`.

The `nowait` parameter is unused. It was part of the 0-9-1 protocol, but this AMQP client implements 0-10 which removed the `nowait` option.

The `arguments` parameter is a set of arguments for the declaration of the queue. Arguments are passed as a dict or `None`. This field is ignored if `passive` is `True`. Default is `None`.

This method returns a `namedtuple` with the name `'queue_declare_ok_t'` and the queue name as `'queue'`, message count on the queue as `'message_count'`, and the number of active consumers as `'consumer_count'`. The named tuple values are ordered as `queue`, `message_count`, and `consumer_count` respectively.

Due to Celery's non-ACKing of events, a ring policy is set on any queue that starts with the string `'celeryev'` or ends with the string `'pidbox'`. These are celery event queues, and Celery does not ack them, causing the messages to build-up. Eventually Qpid stops serving messages unless the `'ring'` policy is set, at which point the buffer backing the queue becomes circular.

#### Parameters

- **queue** (*str*) – The name of the queue to be created.
- **passive** (*bool*) – If True, the sever will not create the queue.
- **durable** (*bool*) – If True, the queue will be durable.
- **exclusive** (*bool*) – If True, the queue will be exclusive.
- **auto\_delete** (*bool*) – If True, the queue is deleted when all consumers have finished using it.
- **nowait** (*bool*) – This parameter is unused since the 0-10 specification does not include it.
- **arguments** (*dict or None*) – A set of arguments for the declaration of the queue.

**Returns** A named tuple representing the declared queue as a named tuple. The tuple values are ordered as `queue`, `message count`, and the active consumer count.

**Return type** `namedtuple`

`Channel.queue_delete(queue, if_unused=False, if_empty=False, **kwargs)`  
Delete a queue by name.

Delete a queue specified by name. Using the `if_unused` keyword argument, the delete can only occur if there are 0 consumers bound to it. Using the `if_empty` keyword argument, the delete can only occur if there are 0 messages in the queue.

#### Parameters

- **queue** (*str*) – The name of the queue to be deleted.
- **if\_unused** (*bool*) – If True, delete only if the queue has 0 consumers. If False, delete a queue even with consumers bound to it.
- **if\_empty** (*bool*) – If True, only delete the queue if it is empty. If False, delete the queue if it is empty or not.

`Channel.queue_purge(queue, **kwargs)`  
Remove all undelivered messages from queue.

Purge all undelivered messages from a queue specified by name. If the queue does not exist an exception is raised. The queue message depth is first checked, and then the broker is asked to purge that number of messages. The integer number of messages requested to be purged is returned. The actual number of messages purged may be different than the requested number of messages to purge.

Sometimes delivered messages are asked to be purged, but are not. This case fails silently, which is the correct behavior when a message that has been delivered to a different consumer, who has not ACKed the message, and still has an active session with the broker. Messages in that case are not safe for purging and will be retained by the broker. The client is unable to change this delivery behavior.

Internally, this method relies on `_purge()`.

**Parameters** `queue` (*str*) – The name of the queue which should have all messages removed.

**Returns** The number of messages requested to be purged.

**Return type** `int`

**Raises** `qpid.messaging.exceptions.NotFound` if the queue being purged cannot be found.

`Channel.queue_unbind` (*queue, exchange, routing\_key, \*\*kwargs*)

Unbind a queue from an exchange with a given bind key.

Unbind a queue specified by name, from an exchange specified by name, that is already bound with a bind key. The queue and exchange must already exist on the broker, and bound with the bind key for the operation to complete successfully. Queues may be bound to exchanges multiple times with different keys, thus the bind key is a required field to unbind in an explicit way.

#### Parameters

- `queue` (*str*) – The name of the queue to be unbound.
- `exchange` (*str*) – The name of the exchange that the queue should be unbound from.
- `routing_key` (*str*) – The existing bind key between the specified queue and a specified exchange that should be unbound.

`Channel.typeof` (*exchange, default='direct'*)

Get the exchange type.

Lookup and return the exchange type for an exchange specified by name. Exchange types are expected to be 'direct', 'topic', and 'fanout', which correspond with exchange functionality as specified in AMQP 0-10 and earlier. If the exchange cannot be found, the default exchange type is returned.

#### Parameters

- `exchange` (*str*) – The exchange to have its type lookup up.
- `default` (*str*) – The type of exchange to assume if the exchange does not exist.

**Returns** The exchange type either 'direct', 'topic', or 'fanout'.

**Return type** `str`

## 4.29.4 Message

`class kombu.transport.qpid.Message` (*channel, payload, \*\*kwargs*)

`serializable()`

## 4.30 kombu.transport.memory

In-memory transport.

- *Transport*
- *Channel*

### 4.30.1 Transport

```
class kombu.transport.memory.Transport (client, **kwargs)
```

```
    class Channel (connection, **kwargs)
```

```
        after_reply_message_received (queue)
```

```
        close ()
```

```
        do_restore = False
```

```
        queues = {}
```

```
        supports_fanout = True
```

```
Transport.driver_name = 'memory'
```

```
Transport.driver_type = 'memory'
```

```
Transport.driver_version ()
```

```
Transport.state = <kombu.transport.virtual.BrokerState object>  
memory backend state is global.
```

### 4.30.2 Channel

```
class kombu.transport.memory.Channel (connection, **kwargs)
```

```
    after_reply_message_received (queue)
```

```
    close ()
```

```
    do_restore = False
```

```
    queues = {}
```

```
    supports_fanout = True
```

## 4.31 kombu.transport.redis

Redis transport.

- *Transport*
- *Channel*

### 4.31.1 Transport

```
class kombu.transport.redis.Transport (*args, **kwargs)
```

```
    class Channel (*args, **kwargs)
```

```
        class QoS (*args, **kwargs)
```

```
            ack (delivery_tag)
```

```
            append (message, delivery_tag)
```

```
            pipe_or_acquire (*args, **kws)
```

```
            reject (delivery_tag, requeue=False)
```

```
            restore_at_shutdown = True
```

```
            restore_by_tag (tag, client=None, leftmost=False)
```

```
            restore_unacked (client=None)
```

```
            restore_visible (start=0, num=10, interval=10)
```

```
            unacked_index_key
```

```
            unacked_key
```

```
            unacked_mutex_expire
```

```
            unacked_mutex_key
```

```
            visibility_timeout
```

```
Transport.Channel.ack_emulation = True
```

```
Transport.Channel.active_queues
```

```
    Set of queues being consumed from (excluding fanout queues).
```

```
Transport.Channel.async_pool
```

```
Transport.Channel.basic_cancel (consumer_tag)
```

```
Transport.Channel.basic_consume (queue, *args, **kwargs)
```

```
Transport.Channel.client
```

```
    Client used to publish messages, BRPOP etc.
```

```
Transport.Channel.close ()
```

```
Transport.Channel.conn_or_acquire (*args, **kws)
```

```
Transport.Channel.fanout_patterns = False
```

```
Transport.Channel.fanout_prefix = False
```

```
Transport.Channel.from_transport_options = (u'body_encoding', u'deadletter_queue', 'ack_emulation')
```

```
Transport.Channel.get_table (exchange)
```

```
Transport.Channel.keyprefix_fanout = '{db}'
```

```
Transport.Channel.keyprefix_queue = '_kombu.binding.%s'
```

```
Transport.Channel.max_connections = 10
```

```
Transport.Channel.pool
Transport.Channel.priority (n)
Transport.Channel.priority_steps = [0, 3, 6, 9]
Transport.Channel.sep = '\x06\x16'
Transport.Channel.socket_connect_timeout = None
Transport.Channel.socket_keepalive = None
Transport.Channel.socket_keepalive_options = None
Transport.Channel.socket_timeout = None
Transport.Channel.subclient
    Pub/Sub connection used to consume fanout queues.
Transport.Channel.supports_fanout = True
Transport.Channel.unacked_index_key = 'unacked_index'
Transport.Channel.unacked_key = 'unacked'
Transport.Channel.unacked_mutex_expire = 300
Transport.Channel.unacked_mutex_key = 'unacked_mutex'
Transport.Channel.unacked_restore_limit = None
Transport.Channel.visibility_timeout = 3600

Transport.default_port = 6379
Transport.driver_name = 'redis'
Transport.driver_type = 'redis'
Transport.driver_version ()
Transport.on_readable (fileno)
    Handle AIO event for one of our file descriptors.
Transport.polling_interval = None
Transport.register_with_event_loop (connection, loop)
Transport.supports_ev = True
```

### 4.31.2 Channel

```
class kombu.transport.redis.Channel (*args, **kwargs)
```

```
    class QoS (*args, **kwargs)

        ack (delivery_tag)
        append (message, delivery_tag)
        pipe_or_acquire (*args, **kwargs)
        reject (delivery_tag, requeue=False)
        restore_at_shutdown = True
```



```

restore_by_tag (tag, client=None, leftmost=False)
restore_unacked (client=None)
restore_visible (start=0, num=10, interval=10)
unacked_index_key
unacked_key
unacked_mutex_expire
unacked_mutex_key
visibility_timeout

Channel.ack_emulation = True

Channel.active_queues
    Set of queues being consumed from (excluding fanout queues).

Channel.async_pool

Channel.basic_cancel (consumer_tag)

Channel.basic_consume (queue, *args, **kwargs)

Channel.client
    Client used to publish messages, BRPOP etc.

Channel.close ()

Channel.conn_or_acquire (*args, **kwargs)

Channel.fanout_patterns = False
    If enabled the fanout exchange will support patterns in routing and binding keys (like a topic exchange but using PUB/SUB). This will be enabled by default in a future version.

Channel.fanout_prefix = False
    Transport option to enable/disable fanout keyprefix. Should be enabled by default, but that is not backwards compatible. Can also be string, in which case it changes the default prefix ('/{db}.') into something else. The prefix must include a leading slash and a trailing dot.

Channel.from_transport_options = ('body_encoding', 'deadletter_queue', 'ack_emulation', 'unacked_key', 'u

Channel.get_table (exchange)

Channel.keyprefix_fanout = '/{db}.'

Channel.keyprefix_queue = '_kombu.binding.%s'

Channel.max_connections = 10

Channel.pool

Channel.priority (n)

Channel.priority_steps = [0, 3, 6, 9]

Channel.sep = '\x06\x16'

Channel.socket_connect_timeout = None

Channel.socket_keepalive = None

Channel.socket_keepalive_options = None

Channel.socket_timeout = None

```

```
Channel.subclient
    Pub/Sub connection used to consume fanout queues.

Channel.supports_fanout = True

Channel.unacked_index_key = 'unacked_index'

Channel.unacked_key = 'unacked'

Channel.unacked_mutex_expire = 300

Channel.unacked_mutex_key = 'unacked_mutex'

Channel.unacked_restore_limit = None

Channel.visibility_timeout = 3600
```

## 4.32 kombu.transport.zmq

### 4.32.1 kombu.transport.zmq

ZeroMQ transport.

```
members
undoc-members
```

## 4.33 kombu.transport.beanstalk

Beanstalk transport.

```
copyright
    3. 2010 - 2013 by David Ziegler.

license BSD, see LICENSE for more details.
```

- *Transport*
- *Channel*

### 4.33.1 Transport

```
class kombu.transport.beanstalk.Transport (*args, **kwargs)
```

```
    class Channel (connection, **kwargs)
```

```
        client
        close ()
```

```
Transport.channel_errors = (<class 'amqp.exceptions.ChannelError'>, <class 'socket.error'>, <type 'exceptions.ImportError'>)
Transport.connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'socket.error'>, <type 'exceptions.ImportError'>)
Transport.default_port = 11300
```

```

Transport.driver_name = 'beanstalkc'
Transport.driver_type = 'beanstalk'
Transport.driver_version()
Transport.polling_interval = 1

```

### 4.33.2 Channel

```
class kombu.transport.beanstalk.Channel (connection, **kwargs)
```

```

client
close()

```

## 4.34 kombu.transport.mongodb

MongoDB transport.

**copyright**

3. 2010 - 2013 by Flavio Percoco Premoli.

**license** BSD, see LICENSE for more details.

- *Transport*
- *Channel*

### 4.34.1 Transport

```
class kombu.transport.mongodb.Transport (client, **kwargs)
```

```
class Channel (*args, **kwargs)
```

```

client
create_broadcast_cursor (exchange, routing_key, pattern, queue)
get_broadcast ()
get_broadcast_cursor (queue)
get_messages ()
get_routing ()
get_table (exchange)
    Get table of bindings for exchange.
queue_delete (queue, **kwargs)
supports_fanout = True
Transport.can_parse_url = True

```

```
Transport.channel_errors = (<class 'amqp.exceptions.ChannelError'>, <class 'pymongo.errors.ConnectionFailure'>)
Transport.connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'pymongo.errors.ConnectionFailure'>)
Transport.default_port = 27017
Transport.driver_name = 'pymongo'
Transport.driver_type = 'mongodb'
Transport.driver_version()
Transport.polling_interval = 1
```

## 4.34.2 Channel

```
class kombu.transport.mongodb.Channel (*vargs, **kwargs)
```

```
    client
    create_broadcast_cursor (exchange, routing_key, pattern, queue)
    get_broadcast ()
    get_broadcast_cursor (queue)
    get_messages ()
    get_routing ()
    get_table (exchange)
        Get table of bindings for exchange.
    queue_delete (queue, **kwargs)
    supports_fanout = True
```

## 4.35 kombu.transport.couchdb

CouchDB transport.

### copyright

3. 2010 - 2013 by David Clymer.

license BSD, see LICENSE for more details.

- *Transport*
- *Channel*
- *Functions*

### 4.35.1 Transport

```
class kombu.transport.couchdb.Transport (*args, **kwargs)
```

```
    class Channel (connection, **kwargs)
```

```

    client
    view_created = False
Transport.channel_errors = (<class 'amqp.exceptions.ChannelError'>, None, None, None, None, None)
Transport.connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'socket.error'>, None, None)
Transport.default_port = 5984
Transport.driver_name = 'couchdb'
Transport.driver_type = 'couchdb'
Transport.driver_version()
Transport.polling_interval = 1

```

## 4.35.2 Channel

```
class kombu.transport.couchdb.Channel(connection, **kwargs)
```

```

    client
    view_created = False

```

## 4.35.3 Functions

```
kombu.transport.couchdb.create_message_view(db)
```

## 4.36 kombu.transport.zookeeper

### 4.36.1 kombu.transport.zookeeper

Zookeeper transport.

#### copyright

3. 2010 - 2013 by Mahendra M.

**license** BSD, see LICENSE for more details.

#### Synopsis

Connects to a zookeeper node as <server>:<port>/<vhost> The <vhost> becomes the base for all the other znodes. So we can use it like a vhost.

This uses the built-in kazoo recipe for queues

#### References

- [https://zookeeper.apache.org/doc/trunk/recipes.html#sc\\_recipes\\_Queues](https://zookeeper.apache.org/doc/trunk/recipes.html#sc_recipes_Queues)
- <https://kazoo.readthedocs.io/en/latest/api/recipe/queue.html>

**Limitations** This queue does not offer reliable consumption. An entry is removed from the queue prior to being processed. So if an error occurs, the consumer has to re-queue the item or it will be lost.

- *Transport*
- *Channel*

## Transport

```
class kombu.transport.zookeeper.Transport (*args, **kwargs)

    class Channel (connection, **kwargs)

        client

Transport.channel_errors = (<class 'amqp.exceptions.ChannelError'>,)
Transport.connection_errors = (<class 'amqp.exceptions.ConnectionError'>,)
Transport.default_port = 2181
Transport.driver_name = 'kazoo'
Transport.driver_type = 'zookeeper'
Transport.driver_version ()
Transport.polling_interval = 1
```

## Channel

```
class kombu.transport.zookeeper.Channel (connection, **kwargs)

    client
```

## 4.37 kombu.transport.filesystem

Transport using the file system as the message store.

- *Transport*
- *Channel*

### 4.37.1 Transport

```
class kombu.transport.filesystem.Transport (client, **kwargs)

    class Channel (connection, **kwargs)

        data_folder_in
        data_folder_out
        processed_folder
```

```

    store_processed
    transport_options
Transport.default_port = 0
Transport.driver_name = 'filesystem'
Transport.driver_type = 'filesystem'
Transport.driver_version()

```

### 4.37.2 Channel

```
class kombu.transport.filesystem.Channel (connection, **kwargs)
```

```

    data_folder_in
    data_folder_out
    processed_folder
    store_processed
    transport_options

```

## 4.38 kombu.transport.django

Kombu transport using the Django database as a message store.

- *Transport*
- *Channel*

### 4.38.1 Transport

```
class kombu.transport.django.Transport (client, **kwargs)
```

```
class Channel (connection, **kwargs)
```

```

    Queue
    basic_consume (queue, *args, **kwargs)
    queue_model = 'kombu.transport.django.models:Queue'
    refresh_connection ()

```

```
Transport.channel_errors = (<class 'amqp.exceptions.ChannelError'>, <class 'django.core.exceptions.ObjectDoesNotExist'>)
```

```

Transport.default_port = 0
Transport.driver_name = 'django'
Transport.driver_type = 'sql'
Transport.driver_version()

```

```
Transport.polling_interval = 5.0
```

## 4.38.2 Channel

```
class kombu.transport.django.Channel (connection, **kwargs)
```

**Queue**

```
basic_consume (queue, *args, **kwargs)
```

```
queue_model = 'kombu.transport.django.models:Queue'
```

```
refresh_connection ()
```

## 4.39 Django Models - kombu.transport.django.models

```
class kombu.transport.django.models.Message (id, visible, sent_at, payload, queue)
```

**exception DoesNotExist**

**exception Message.MultipleObjectsReturned**

**Message.id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**Message.objects = <kombu.transport.django.managers.MessageManager object>**

**Message.payload**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**Message.queue**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child (Model) :
    parent = ForeignKey (Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

**Message.queue\_id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**Message.sent\_at**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**Message.visible**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

```
class kombu.transport.django.models.Queue (id, name)
```

**exception DoesNotExist**



**exception Queue.MultipleObjectsReturned****Queue.id**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

**Queue.messages**

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

**Queue.name**

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

`Queue.objects = <kombu.transport.django.managers.QueueManager object>`

## 4.40 Django Managers - kombu.transport.django.managers

```
class kombu.transport.django.managers.MessageManager
```

```
    cleanup()
```

```
    cleanup_every = 10
```

```
    connection_for_write()
```

```
    pop(*args, **kwargs)
```

```
class kombu.transport.django.managers.QueueManager
```

```
    fetch(queue_name)
```

```
    publish(queue_name, payload)
```

```
    purge(queue_name)
```

```
    size(queue_name)
```

```
kombu.transport.django.managers.commit_on_success(fun)
```

```
kombu.transport.django.managers.select_for_update(qs)
```

## 4.41 Django Management - clean\_kombu\_messages

members

undoc-members

## 4.42 kombu.transport.sqlalchemy

## 4.43 kombu.transport.SLMQ

### 4.43.1 kombu.transport.SLMQ

SoftLayer Message Queue transport.

- *Transport*
- *Channel*

#### Transport

```
class kombu.transport.SLMQ.Transport(client, **kwargs)
```

```
    class Channel(*args, **kwargs)
```

```
        basic_ack(delivery_tag)
```

```
        basic_cancel(consumer_tag)
```

```
        basic_consume(queue, no_ack, *args, **kwargs)
```

```
        conninfo
```

```
        default_visibility_timeout = 1800
```

```
        delete_message(queue, message_id)
```

```
        domain_format = 'kombu%(vhost)s'
```

```
        entity_name(name, table={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 45: 95, 46: 95, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95})
```

```
            Format AMQP queue name into a valid SLQS queue name.
```

```
        queue_name_prefix
```

```
        slmq
```

```
        transport_options
```

```
        visibility_timeout
```

```
Transport.connection_errors = (<class 'amqp.exceptions.ConnectionError'>, None, <class 'socket.error'>)
```

```
Transport.default_port = None
```

```
Transport.polling_interval = 1
```

#### Channel

```
class kombu.transport.SLMQ.Channel(*args, **kwargs)
```

```

basic_ack (delivery_tag)
basic_cancel (consumer_tag)
basic_consume (queue, no_ack, *args, **kwargs)
conninfo
default_visibility_timeout = 1800
delete_message (queue, message_id)
domain_format = 'kombu%(vhost)s'
entity_name (name, table={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95,
42: 95, 43: 95, 44: 95, 45: 95, 46: 95, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95,
63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126:
95})
    Format AMQP queue name into a valid SLQS queue name.
queue_name_prefix
slmq
transport_options
visibility_timeout

```

## 4.44 kombu.transport.pyro

Pyro transport.

Requires the Pyro4 library to be installed.

- *Transport*
- *Channel*

### 4.44.1 Transport

```
class kombu.transport.pyro.Transport (client, **kwargs)
```

```
    class Channel (connection, **kwargs)
```

```
        after_reply_message_received (queue)
```

```
        queues ()
```

```
        shared_queues
```

```
Transport.default_port = 9090
```

```
Transport.driver_name = 'pyro'
```

```
Transport.driver_type = 'pyro'
```

```
Transport.driver_version ()
```

```
Transport.shared_queues
```

`Transport.state = <kombu.transport.virtual.BrokerState object>`  
memory backend state is global.

## 4.44.2 Channel

```
class kombu.transport.pyro.Channel (connection, **kwargs)
```

```
    after_reply_message_received (queue)
```

```
    queues ()
```

```
    shared_queues
```

## 4.45 kombu.transport.amqplib

amqplib transport.

- *Transport*
- *Connection*
- *Channel*
- *Message*

### 4.45.1 Transport

```
class kombu.transport.amqplib.Transport (client, **kwargs)
```

```
    class Connection (*args, **kwargs)
```

```
        channel (channel_id=None)
```

```
        connected = True
```

```
        drain_events (timeout=None)
```

```
            Wait for an event on a channel.
```

```
        read_timeout (timeout=None)
```

```
Transport.channel_errors = (<class 'amqp.exceptions.ChannelError'>, <class 'kombu.transport.amqplib.NA'>)
```

```
Transport.close_connection (connection)
```

```
    Close the AMQP broker connection.
```

```
Transport.connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'kombu.transport.amqplib.N'>)
```

```
Transport.create_channel (connection)
```

```
Transport.default_connection_params
```

```
Transport.default_port = 5672
```

```
Transport.drain_events (connection, **kwargs)
```

```
Transport.driver_name = 'amqplib'
```

```

Transport.driver_type = 'amqp'
Transport.establish_connection()
    Establish connection to the AMQP broker.
Transport.get_manager(*args, **kwargs)
Transport.is_alive(connection)
Transport.register_with_event_loop(connection, loop)
Transport.supports_ev = True
Transport.verify_connection(connection)

```

## 4.45.2 Connection

```
class kombu.transport.amqplib.Connection(*args, **kwargs)
```

```
    AMQP_PROTOCOL_HEADER = 'AMQP\x01\x01\x08\x00'
```

```
    Connection
        alias of NA
```

```
class SSLTransport(host, connect_timeout, ssl)
```

```
    read_frame()
```

```
class Connection.TCPTransport
```

```
    read_frame()
```

```
Connection.channel(channel_id=None)
```

```
Connection.connected = True
```

```
Connection.drain_events(timeout=None)
    Wait for an event on a channel.
```

```
Connection.read_timeout(timeout=None)
```

## 4.45.3 Channel

```
class kombu.transport.amqplib.Channel(*args, **kwargs)
```

```
class Message(channel, msg, **kwargs)
```

```
Channel.basic_cancel(consumer_tag, **kwargs)
```

```
Channel.basic_consume(*args, **kwargs)
```

```
Channel.close()
```

```
Channel.events = {'basic_return': set({})}
```

```
Channel.message_to_python(raw_message)
    Convert encoded message body back to a Python value.
```

`Channel.prepare_message` (*body*, *priority=None*, *content\_type=None*, *content\_encoding=None*,  
*headers=None*, *properties=None*)  
Encapsulate data into a AMQP message.

#### 4.45.4 Message

```
class kombu.transport.amqplib.Message (channel, msg, **kwargs)
```

### 4.46 kombu.transport.base

Base transport interface.

- *Message*
- *Transport*

#### 4.46.1 Message

```
class kombu.transport.base.Message (channel, body=None, delivery_tag=None, content_type=None, content_encoding=None, delivery_info={}, properties=None, headers=None, postencode=None, accept=None, **kwargs)
```

Base class for received messages.

**payload**

The decoded message body.

**channel**

**delivery\_tag**

**content\_type**

**content\_encoding**

**delivery\_info**

**headers**

**properties**

**body**

**acknowledged**

Set to true if the message has been acknowledged.

**ack ()**

Acknowledge this message as being processed., This will remove the message from the queue.

**Raises MessageStateError** – If the message has already been acknowledged/requeued/rejected.

**reject (requeue=False)**

Reject this message.

The message will be discarded by the server.

**Raises `MessageStateError`** – If the message has already been acknowledged/requeued/rejected.

**`requeue ()`**

Reject this message and put it back on the queue.

You must not use this method as a means of selecting messages to process.

**Raises `MessageStateError`** – If the message has already been acknowledged/requeued/rejected.

**`decode ()`**

Deserialize the message body, returning the original python structure sent by the publisher.

## 4.46.2 Transport

**class `kombu.transport.base.Transport`** (*client*, *\*\*kwargs*)

Base class for transports.

**`client = None`**

The `Connection` owning this instance.

**`default_port = None`**

Default port used when no port has been specified.

**`recoverable_connection_errors`**

Optional list of connection related exceptions that can be recovered from, but where the connection must be closed and re-established first.

If not defined then all `connection_errors` and `channel_errors` will be regarded as recoverable, but needing to close the connection first.

**`recoverable_channel_errors`**

Optional list of channel related exceptions that can be automatically recovered from without re-establishing the connection.

**`connection_errors = (<class 'amqp.exceptions.ConnectionError'>,)`**

Tuple of errors that can happen due to connection failure.

**`channel_errors = (<class 'amqp.exceptions.ChannelError'>,)`**

Tuple of errors that can happen due to channel/method failure.

**`establish_connection ()`**

**`close_connection (connection)`**

**`create_channel (connection)`**

**`close_channel (connection)`**

**`drain_events (connection, **kwargs)`**

## 4.47 kombu.transport.virtual

Virtual transport implementation.

Emulates the AMQ API for non-AMQ transports.

- *Transports*
- *Channel*
- *Message*
- *Quality Of Service*
- *In-memory State*

### 4.47.1 Transports

```
class kombu.transport.virtual.Transport (client, **kwargs)
    Virtual transport.
        Parameters client – Connection instance
        Channel = <class 'kombu.transport.virtual.Channel'>
        Cycle = <class 'kombu.transport.virtual.scheduling.FairCycle'>
        polling_interval = 1.0
            Time to sleep between unsuccessful polls.
        default_port = None
            port number used when no port is specified.
        state = <kombu.transport.virtual.BrokerState object>
            BrokerState containing declared exchanges and bindings (set by constructor).
        cycle = None
            FairCycle instance used to fairly drain events from channels (set by constructor).
        establish_connection ()
        close_connection (connection)
        create_channel (connection)
        close_channel (channel)
        drain_events (connection, timeout=None)
```

### 4.47.2 Channel

```
class kombu.transport.virtual.AbstractChannel
    This is an abstract class defining the channel methods you'd usually want to implement in a virtual channel.
    Do not subclass directly, but rather inherit from Channel instead.
class kombu.transport.virtual.Channel (connection, **kwargs)
    Virtual channel.
        Parameters connection – The transport instance this channel is part of.
        Message = <class 'kombu.transport.virtual.Message'>
            message class used.
        state
            Broker state containing exchanges and bindings.
        qos
            QoS manager for this channel.
        do_restore = True
            flag to restore unacked messages when channel goes out of scope.
```



**exchange\_types** = {'topic': <class 'kombu.transport.virtual.exchange.TopicExchange'>, 'fanout': <class 'kombu.trans...>  
mapping of exchange types and corresponding classes.

**exchange\_declare** (*exchange=None, type=u'direct', durable=False, auto\_delete=False, arguments=None, nowait=False, passive=False*)  
Declare exchange.

**exchange\_delete** (*exchange, if\_unused=False, nowait=False*)  
Delete *exchange* and all its bindings.

**queue\_declare** (*queue=None, passive=False, \*\*kwargs*)  
Declare queue.

**queue\_delete** (*queue, if\_unused=False, if\_empty=False, \*\*kwargs*)  
Delete queue.

**queue\_bind** (*queue, exchange=None, routing\_key=u'', arguments=None, \*\*kwargs*)  
Bind *queue* to *exchange* with *routing key*.

**queue\_purge** (*queue, \*\*kwargs*)  
Remove all ready messages from queue.

**basic\_publish** (*message, exchange, routing\_key, \*\*kwargs*)  
Publish message.

**basic\_consume** (*queue, no\_ack, callback, consumer\_tag, \*\*kwargs*)  
Consume from *queue*

**basic\_cancel** (*consumer\_tag*)  
Cancel consumer by consumer tag.

**basic\_get** (*queue, no\_ack=False, \*\*kwargs*)  
Get message by direct access (synchronous).

**basic\_ack** (*delivery\_tag*)  
Acknowledge message.

**basic\_recover** (*requeue=False*)  
Recover unacked messages.

**basic\_reject** (*delivery\_tag, requeue=False*)  
Reject message.

**basic\_qos** (*prefetch\_size=0, prefetch\_count=0, apply\_global=False*)  
Change QoS settings for this channel.  
Only *prefetch\_count* is supported.

**get\_table** (*exchange*)  
Get table of bindings for *exchange*.

**typeof** (*exchange, default=u'direct'*)  
Get the exchange type instance for *exchange*.

**drain\_events** (*timeout=None*)

**prepare\_message** (*body, priority=None, content\_type=None, content\_encoding=None, headers=None, properties=None*)  
Prepare message data.

**message\_to\_python** (*raw\_message*)  
Convert raw message to *Message* instance.

**flow** (*active=True*)  
Enable/disable message flow.

Raises `NotImplementedError` – as flow is not implemented by the base virtual implementation.

`close()`

Close channel, cancel all consumers, and requeue unacked messages.

### 4.47.3 Message

`class kombu.transport.virtual.Message(channel, payload, **kwargs)`

**exception MessageStateError**

The message has already been acknowledged.

**args**

**message**

`Message.accept`

`Message.ack()`

Acknowledge this message as being processed., This will remove the message from the queue.

Raises `MessageStateError` – If the message has already been acknowledged/requeued/rejected.

`Message.ack_log_error(logger, errors)`

`Message.acknowledged`

Set to true if the message has been acknowledged.

`Message.body`

`Message.channel`

`Message.content_encoding`

`Message.content_type`

`Message.decode()`

Deserialize the message body, returning the original python structure sent by the publisher.

`Message.delivery_info`

`Message.delivery_tag`

`Message.errors = None`

`Message.headers`

`Message.payload`

The decoded message body.

`Message.properties`

`Message.reject(rewrite=False)`

Reject this message.

The message will be discarded by the server.

Raises `MessageStateError` – If the message has already been acknowledged/requeued/rejected.

`Message.reject_log_error(logger, errors, rewrite=False)`

`Message.requeue()`

Reject this message and put it back on the queue.

You must not use this method as a means of selecting messages to process.

**Raises `MessageStateError`** – If the message has already been acknowledged/requeued/rejected.

`Message.serializable()`

#### 4.47.4 Quality Of Service

`class kombu.transport.virtual.QoS(channel, prefetch_count=0)`

Quality of Service guarantees.

Only supports `prefetch_count` at this point.

**Parameters**

- **channel** – AMQ Channel.
- **prefetch\_count** – Initial prefetch count (defaults to 0).

`ack(delivery_tag)`

Acknowledge message and remove from transactional state.

`append(message, delivery_tag)`

Append message to transactional state.

`can_consume()`

Return true if the channel can be consumed from.

Used to ensure the client adheres to currently active prefetch limits.

`can_consume_max_estimate()`

Returns the maximum number of messages allowed to be returned.

Returns an estimated number of messages that a consumer may be allowed to consume at once from the broker. This is used for services where bulk ‘get message’ calls are preferred to many individual ‘get message’ calls - like SQS.

**returns:** An integer > 0

`get(delivery_tag)`

`prefetch_count = 0`

current prefetch count value

`reject(delivery_tag, requeue=False)`

Remove from transactional state and requeue message.

`restore_at_shutdown = True`

If disabled, unacked messages won’t be restored at shutdown.

`restore_unacked()`

Restore all unacknowledged messages.

`restore_unacked_once()`

Restores all unacknowledged messages at shutdown/gc collect.

Will only be done once for each instance.

`restore_visible(*args, **kwargs)`

Restore any pending unacknowledged messages for `visibility_timeout` style implementations.

Optional: Currently only used by the Redis transport.

### 4.47.5 In-memory State

**class** kombu.transport.virtual.**BrokerState** (*exchanges=None, bindings=None*)

```
bindings = None
    active bindings.

clear ()

exchanges = None
    exchange declarations.
```

## 4.48 kombu.transport.virtual.exchange

Implementations of the standard exchanges defined by the AMQ protocol (excluding the *headers* exchange).

- *Direct*
- *Topic*
- *Fanout*
- *Interface*

### 4.48.1 Direct

**class** kombu.transport.virtual.exchange.**DirectExchange** (*channel*)

The *direct* exchange routes based on exact routing keys.

```
deliver (message, exchange, routing_key, **kwargs)
```

```
lookup (table, exchange, routing_key, default)
```

```
type = 'direct'
```

### 4.48.2 Topic

**class** kombu.transport.virtual.exchange.**TopicExchange** (*channel*)

The *topic* exchange routes messages based on words separated by dots, using wildcard characters \* (any single word), and # (one or more words).

```
deliver (message, exchange, routing_key, **kwargs)
```

```
key_to_pattern (rkey)
```

Get the corresponding regex for any routing key.

```
lookup (table, exchange, routing_key, default)
```

```
prepare_bind (queue, exchange, routing_key, arguments)
```

```
type = 'topic'
```

```
wildcards = {'#': '.*?', '*': '.*?[^\.]'}
```

map of wildcard to regex conversions

### 4.48.3 Fanout

**class** `kombu.transport.virtual.exchange.FanoutExchange` (*channel*)

The *fanout* exchange implements broadcast messaging by delivering copies of all messages to all queues bound to the exchange.

To support fanout the virtual channel needs to store the table as shared state. This requires that the *Channel.supports\_fanout* attribute is set to true, and the *Channel.\_queue\_bind* and *Channel.get\_table* methods are implemented. See the redis backend for an example implementation of these methods.

**deliver** (*message, exchange, routing\_key, \*\*kwargs*)

**lookup** (*table, exchange, routing\_key, default*)

**type** = 'fanout'

### 4.48.4 Interface

**class** `kombu.transport.virtual.exchange.ExchangeType` (*channel*)

Implements the specifics for an exchange type.

**Parameters** **channel** – AMQ Channel

**equivalent** (*prev, exchange, type, durable, auto\_delete, arguments*)

Return true if *prev* and *exchange* is equivalent.

**lookup** (*table, exchange, routing\_key, default*)

Lookup all queues matching *routing\_key* in *exchange*.

**Returns** *default* if no queues matched.

**prepare\_bind** (*queue, exchange, routing\_key, arguments*)

Return tuple of (*routing\_key, regex, queue*) to be stored for bindings to this exchange.

**type** = None

- `kombu.transport.virtual.scheduling`

## 4.49 kombu.transport.virtual.scheduling

Consumer utilities.

**class** `kombu.transport.virtual.scheduling.FairCycle` (*fun, resources, predicate=<type 'exceptions.Exception'>*)

Consume from a set of resources, where each resource gets an equal chance to be consumed from.

**close** ()

**get** (*\*\*kwargs*)

## 4.50 kombu.serialization

Serialization utilities.

- [Overview](#)
- [Exceptions](#)
- [Serialization](#)
- [Registry](#)

### 4.50.1 Overview

Centralized support for encoding/decoding of data structures. Contains json, pickle, msgpack, and yaml serializers.

Optionally installs support for YAML if the [PyYAML](#) package is installed.

Optionally installs support for [msgpack](#) if the [msgpack-python](#) package is installed.

### 4.50.2 Exceptions

**exception** `kombu.serialization.SerializerNotInstalled`

Support for the requested serialization type is not installed

### 4.50.3 Serialization

`kombu.serialization.encode` (*self*, *data*, *serializer=None*)

**loads** (*data*, *content\_type*, *content\_encoding*):

Deserialize a data stream as serialized using *dumps* based on *content\_type*.

**Parameters**

- **data** – The message data to deserialize.
- **content\_type** – The content-type of the data. (e.g., *application/json*).
- **content\_encoding** – The content-encoding of the data. (e.g., *utf-8*, *binary*, or *us-ascii*).

**Returns** The unserialized data.

`kombu.serialization.decode` (*self*, *data*, *content\_type*, *content\_encoding*, *accept=None*, *force=False*, *\_trusted\_content=frozenset(['application/data', 'application/text'])*)

**register** (*name*, *encoder*, *decoder*, *content\_type*, *content\_encoding='utf-8'*):

Register a new encoder/decoder.

**Parameters**

- **name** – A convenience name for the serialization method.
- **encoder** – A method that will be passed a python data structure and should return a string representing the serialized data. If `None`, then only a decoder will be registered. Encoding will not be possible.
- **decoder** – A method that will be passed a string representing serialized data and should return a python data structure. If `None`, then only an encoder will be registered. Decoding will not be possible.
- **content\_type** – The mime-type describing the serialized structure.

- **content\_encoding** – The content encoding (character set) that the *decoder* method will be returning. Will usually be *utf-8*, *us-ascii*, or *binary*.

`kombu.serialization.raw_encode(data)`

Special case serializer.

#### 4.50.4 Registry

`kombu.serialization.register(self, name, encoder, decoder, content_type, content_encoding='utf-8')`

**unregister(name) :**

Unregister registered encoder/decoder.

**Parameters** *name* – Registered serialization method name.

`kombu.serialization.registry = <kombu.serialization.SerializerRegistry object>`

`kombu.serialization.dumps(data, serializer=default_serializer)`

Serialize a data structure into a string suitable for sending as an AMQP message body.

**Parameters**

- **data** – The message data to send. Can be a list, dictionary or a string.
- **serializer** – An optional string representing the serialization method you want the data marshalled into. (For example, *json*, *raw*, or *pickle*).

If `None` (default), then *json* will be used, unless *data* is a `str` or `unicode` object. In this latter case, no serialization occurs as it would be unnecessary.

Note that if *serializer* is specified, then that serialization method will be used even if a `str` or `unicode` object is passed in.

**Returns** A three-item tuple containing the content type (e.g., *application/json*), content encoding, (e.g., *utf-8*) and a string containing the serialized data.

**Raises** `SerializerNotInstalled` – If the serialization method requested is not available.

## 4.51 Utilities - kombu.utils

- *kombu.utils*

### 4.51.1 kombu.utils

Internal utilities.

**class** `kombu.utils.EqualityDict`

`kombu.utils.say(m, *fargs, **fkwards)`

`kombu.utils.uuid()`

Generate a unique id, having - hopefully - a very small chance of collision.

For now this is provided by `uuid.uuid4()`.

`kombu.utils.kwdict` (*kwargs*)

`kombu.utils.maybe_list` (*v*)

`kombu.utils.fxrange` (*start=1.0, stop=None, step=1.0, repeatlast=False*)

`kombu.utils.fxrangemax` (*start=1.0, stop=None, step=1.0, max=100.0*)

`kombu.utils.retry_over_time` (*fun, catch, args=[], kwargs={}, errback=None, max\_retries=None, interval\_start=2, interval\_step=2, interval\_max=30, callback=None*)

Retry the function over and over until max retries is exceeded.

For each retry we sleep a for a while before we try again, this interval is increased for every retry until the max seconds is reached.

#### Parameters

- **fun** – The function to try
- **catch** – Exceptions to catch, can be either tuple or a single exception class.
- **args** – Positional arguments passed on to the function.
- **kwargs** – Keyword arguments passed on to the function.
- **errback** – Callback for when an exception in `catch` is raised. The callback must take two arguments: `exc` and `interval`, where `exc` is the exception instance, and `interval` is the time in seconds to sleep next..
- **max\_retries** – Maximum number of retries before we give up. If this is not set, we will retry forever.
- **interval\_start** – How long (in seconds) we start sleeping between retries.
- **interval\_step** – By how much the interval is increased for each retry.
- **interval\_max** – Maximum number of seconds to sleep between retries.

`kombu.utils.emergency_dump_state` (*state, open\_file=<built-in function open>, dump=None*)

**class** `kombu.utils.cached_property` (*fget=None, fset=None, fdel=None, doc=None*)

Property descriptor that caches the return value of the get function.

#### Examples

```
@cached_property
def connection(self):
    return Connection()

@connection.setter # Prepares stored value
def connection(self, value):
    if value is None:
        raise TypeError('Connection must be a connection')
    return value

@connection.deleter
def connection(self, value):
    # Additional action to do at del(self.attr)
    if value is not None:
        print('Connection {0!r} deleted'.format(value))
```

**deleter** (*fdel*)

**setter** (*fset*)



`kombu.utils.reprkwargs` (*kwargs*, *sep*=' ', *fmt*='{0}={1}')

`kombu.utils.reprcall` (*name*, *args*=(), *kwargs*={}, *sep*=' ', '')

`kombu.utils.nested` (*\*args*, *\*\*kws*)

Combine multiple context managers into a single nested context manager.

`kombu.utils.fileno` (*f*)

`kombu.utils.maybe_fileno` (*f*)

Get object fileno, or None if not defined.

## 4.52 Evented I/O - kombu.utils.eventio

- [\*kombu.utils.eventio\*](#)

### 4.52.1 kombu.utils.eventio

Evented IO support for multiple platforms.

`kombu.utils.eventio.poll` (*\*args*, *\*\*kwargs*)

## 4.53 Rate limiting - kombu.utils.limits

- [\*kombu.utils.limits\*](#)

### 4.53.1 kombu.utils.limits

Token bucket implementation for rate limiting.

**class** `kombu.utils.limits.TokenBucket` (*fill\_rate*, *capacity*=1)

Token Bucket Algorithm.

See [http://en.wikipedia.org/wiki/Token\\_Bucket](http://en.wikipedia.org/wiki/Token_Bucket) Most of this code was stolen from an entry in the ASPN Python Cookbook: <http://code.activestate.com/recipes/511490/>

---

#### Thread safety

This implementation is not thread safe. Access to a *TokenBucket* instance should occur within the critical section of any multithreaded code.

---

**can\_consume** (*tokens*=1)

Return `True` if the number of tokens can be consumed from the bucket. If they can be consumed, a call will also consume the requested number of tokens from the bucket. Calls will only consume *tokens* (the number requested) or zero tokens – it will never consume a partial number of tokens.

**capacity** = 1

Maximum number of tokens in the bucket.

**expected\_time** (*tokens=1*)

Return the time (in seconds) when a new token is expected to be available. This will not consume any tokens from the bucket.

**fill\_rate** = None

The rate in tokens/second that the bucket will be refilled.

**timestamp** = None

Timestamp of the last time a token was taken out of the bucket.

## 4.54 Compat. utilities - kombu.utils.compat

- *kombu.utils.compat*

### 4.54.1 kombu.utils.compat

Helps compatibility with older Python versions.

`kombu.utils.compat.get_errno(exc)`

`socket.error` and `IOError` first got the `.errno` attribute in Py2.7

`kombu.utils.compat.timedelta_seconds(delta)`

Convert `datetime.timedelta` to seconds.

Doesn't account for negative values.

## 4.55 Debugging - kombu.utils.debug

- *kombu.utils.debug*

### 4.55.1 kombu.utils.debug

Debugging support.

`kombu.utils.debug.setup_logging(loglevel=10, loggers=['kombu.connection', 'kombu.channel'])`

`class kombu.utils.debug.Logwrapped(instance, logger=None, ident=None)`

## 4.56 String Encoding - kombu.utils.encoding

- *kombu.utils.encoding*

### 4.56.1 kombu.utils.encoding

Utilities to encode text, and to safely emit text from running applications without crashing with the infamous `UnicodeDecodeError` exception.

`kombu.utils.encoding.bytes_to_str(s)`

`kombu.utils.encoding.default_encode(obj, file=None)`

`kombu.utils.encoding.default_encoding(file=None)`

`kombu.utils.encoding.default_encoding_file = None`

`safe_str` takes encoding from this file by default. `set_default_encoding_file()` can be used to set the default output file.

`kombu.utils.encoding.ensure_bytes(s)`

`kombu.utils.encoding.from_utf8(s, *args, **kwargs)`

`kombu.utils.encoding.get_default_encoding_file()`

`kombu.utils.encoding.safe_repr(o, errors='replace')`

`kombu.utils.encoding.safe_str(s, errors='replace')`

`kombu.utils.encoding.set_default_encoding_file(file)`

`kombu.utils.encoding.str_to_bytes(s)`

## 4.57 kombu.utils.functional

**class** `kombu.utils.functional.lazy(fun, *args, **kwargs)`

Holds lazy evaluation.

Evaluated when called or if the `evaluate()` method is called. The function is re-evaluated on every call.

**Overloaded operations that will evaluate the promise:** `__str__()`, `__repr__()`, `__cmp__()`.

`evaluate()`

`kombu.utils.functional.maybe_evaluate(value)`

Evaluates if the value is a `lazy` instance.

`kombu.utils.functional.is_list(l, scalars=(<class 'abcoll.Mapping'>, <type 'basestring'>),  
iters=(<class 'abcoll.Iterable'>,))`

Return true if the object is iterable (but not if object is a mapping or string).

`kombu.utils.functional.maybe_list(l, scalars=(<class 'abcoll.Mapping'>, <type 'basestring'>))`

Return list of one element if `l` is a scalar.

## 4.58 kombu.utils.url

`kombu.utils.url.as_url(scheme, host=None, port=None, user=None, password=None, path=None,  
query=None, sanitize=False, mask='**')`

`kombu.utils.url.maybe_sanitize_url(url, mask='**')`

`kombu.utils.url.parse_url(url)`

`kombu.utils.url.sanitize_url(url, mask='**')`

## 4.59 Text utilites - kombu.utils.text

`kombu.utils.text.fmatch_best` (*needle, haystack, min\_ratio=0.6*)

`kombu.utils.text.fmatch_iter` (*needle, haystack, min\_ratio=0.6*)

`kombu.utils.text.version_string_as_tuple` (*s*)

## 4.60 Generic RabbitMQ manager - kombu.utils.amq\_manager

`kombu.utils.amq_manager.get_manager` (*client, hostname=None, port=None, userid=None, password=None*)

## 4.61 Python2 to Python3 utilities - kombu.five

- *celery.five*

### 4.61.1 celery.five

Compatibility implementations of features only available in newer Python versions.

`kombu.five.exec_` (*code, globs=None, locs=None*)

Execute code in a namespace.

`kombu.five.items` (*d*)

`kombu.five.keys` (*d*)

`kombu.five.monotonic` ()

`kombu.five.nextfun` (*it*)

`kombu.five.reraise` (*tp, value, tb=None*)

**class** `kombu.five.timespec`

**tv\_nsec**

Structure/Union member

**tv\_sec**

Structure/Union member

`kombu.five.values` (*d*)

`kombu.five.with_metaclass` (*Type, skip\_attrs=set(['\_\_dict\_\_', '\_\_weakref\_\_'])*)

Class decorator to set metaclass.

Works with both Python 2 and Python 3 and it does not add an extra class in the lookup order like `six.with_metaclass` does (that is – it copies the original class instead of using inheritance).

---

## Change history

---

### 5.1 3.0.37

**release-date** 2016-10-06 05:00 P.M PDT

**release-by** Ask Solem

- Connection: Return value of `.info()` was no longer JSON serializable, leading to “itertools.cycle object not JSON serializable” errors (Issue #635).

### 5.2 3.0.36

**release-date** 2016-09-30 03:06 P.M PDT

**release-by** Ask Solem

- Connection: Fixed bug when cloning connection with alternate urls.  
Fix contributed by Emmanuel Cazenave.
- Redis: Fixed problem with unix socket connections.  
<https://github.com/celery/celery/issues/2903>  
Fix contributed by Raphael Michel.
- Redis: Fixed compatibility with older redis-py versions (Issue #576).
- Broadcast now retains queue name when being copied/pickled (Issue #578).

### 5.3 3.0.35

**release-date** 2016-03-22 11:22 P.M PST

**release-by** Ask Solem

- msgpack: msgpack support now requires msgpack-python > 0.4.7.
- Redis: TimeoutError was no longer handled as a recoverable error.
- Redis: Adds the ability to set more Redis connection options using `Connection(transport_options={...})`.
  - `socket_connect_timeout` (requires redis-py 2.10 or later)

- `socket_keepalive` (requires `redis-py` 2.10 or later)
- `socket_keepalive_options` (requires `redis-py` 2.10 or later)
- `msgpack`: Fixes support for binary/unicode data

## 5.4 3.0.34

**release-date** 2016-03-03 05:30 P.M PST

**release-by** Ask Solem

- `Qpid`: Adds async error handling.  
Contributed by Brian Bouterse.
- `Qpid`: Delivery tag is now a UUID4 (Issue #563).  
Fix contributed by Brian Bouterse.
- `Redis`: `Connection.as_uri()` returned malformed URLs when the `redis+socket` scheme was used (Issue `celery/celery#2995`).
- `msgpack`: Use binary encoding instead of utf-8 (Issue #570).

## 5.5 3.0.33

**release-date** 2016-01-08 06:36 P.M PST

**release-by** Ask Solem

- Now depends on `amqp` 1.4.9.
- `Redis`: Fixed problem with auxilliary connections causing the main consumer connection to be closed (Issue #550).
- `Qpid`: No longer uses threads to operate, to ensure compatibility with all environments (Issue #531).

## 5.6 3.0.32

**release-date** 2015-12-16 02:29 P.M PST

**release-by** Ask Solem

- `Redis`: Fixed bug introduced in 3.0.31 where the `redis` transport always connects to `localhost`, regardless of host setting.

## 5.7 3.0.31

**release-date** 2015-12-16 12:00 P.M PST

**release-by** Ask Solem

- `Redis`: Fixed bug introduced in 3.0.30 where `socket` was prematurely disconnected.
- `Hub`: Removed debug logging message: “Deregistered fd...” (Issue #549).

## 5.8 3.0.30

**release-date** 2015-12-07 12:28 A.M PST

**release-by** Ask Solem

- Fixes compatibility with uuid in Python 2.7.11 and 3.5.1.  
Fix contributed by Kai Groner.
- Redis transport: Attempt at fixing problem with hanging consumer after disconnected from server.
- **Event loop:** Attempt at fixing issue with 100% CPU when using the Redis transport,
- Database transport: Fixed oracle compatibility.  
An “ORA-00907: missing right parenthesis” error could manifest when using an Oracle database with the database transport.  
Fix contributed by Deepak N.
- Documentation fixes  
Contributed by Tommaso Barbugli.

## 5.9 3.0.29

**release-date** 2015-10-26 11:10 A.M PDT

**release-by** Ask Solem

- Fixed serialization issue for `bindings.as_dict()` (Issue #453).  
Fix contributed by Sergey Tikhonov.
- Json serializer wrongly treated bytes as `ascii`, not `utf-8` (Issue #532).
- MongoDB: Now supports pymongo 3.x.  
Contributed by Len Buckens.
- SQS: Tests passing on Python 3.  
Fix contributed by Felix Yan

## 5.10 3.0.28

**release-date** 2015-10-12 12:00 P.M PDT

**release-by** Ask Solem

.. admonition:: Django transport migrations.

If you’re using Django 1.8 and have already created the `kombu_transport_django` tables, you have to run a fake initial migration:

```
python manage.py migrate kombu_transport_django --fake-initial
```

- No longer compatible with South by default.

To keep using `kombu.transport.django` with South migrations you now need to configure a new location for the kombu migrations:

```
SOUTH_MIGRATION_MODULES = {
    'kombu_transport_django':
        'kombu.transport.django.south_migrations',
}
```

- Keep old South migrations in `kombu.south.migrations`.
- Now works with Redis < 2.10 again.

### 5.11 3.0.27

**release-date** 2015-10-09 3:10 P.M PDT

**release-by** Ask Solem

- Now depends on `amqp` 1.4.7.
- Fixed `libSystem` import error on some OS X 10.11 (El Capitan) installations.  
Fix contributed by Eric Wang.
- Now compatible with Django 1.9.
- Django: Adds migrations for the database transport.
- Redis: Now depends on `py-redis` 2.10.0 or later (Issue #468).
- QPid: Can now connect as `localhost` (Issue #519).  
Fix contributed by Brian Bouterse.
- QPid: Adds support for `login_method` (Issue #502, Issue #499).  
Contributed by Brian Bouterse.
- QPid: Now reads SASL mechanism from broker string (Issue #498).  
Fix contributed by Brian Bouterse.
- QPid: Monitor thread now properly terminated on session close (Issue #485).  
Fix contributed by Brian Bouterse.
- QPid: Fixed file descriptor leak (Issue #476).  
Fix contributed by Jeff Ortel
- Docs: Fixed wrong order for endpoint arguments (Issue #473).
- ConsumerMixin: Connection error logs now include traceback (Issue #480).
- BaseTransport now raises `RecoverableConnectionError` when disconnected (Issue #507).
- Consumer: Adds `tag_prefix` option to modify how consumer tags are generated (Issue #509).

### 5.12 3.0.26

**release-date** 2015-04-22 06:00 P.M UTC

**release-by** Ask Solem



- Fixed compatibility with py-redis versions before 2.10.3 (Issue #470).

## 5.13 3.0.25

**release-date** 2015-04-21 02:00 P.M UTC

**release-by** Ask Solem

- `pyamqp/librabbitmq` now uses 5671 as default port when SSL is enabled (Issue #459).
- Redis: Now supports passwords in `redis+socket://:pass@host:port` URLs (Issue #460).
- `Producer.publish` now defines the `expiration` property in support of the RabbitMQ per-message TTL extension.

Contributed by Anastasis Andronidis.

- Connection transport attribute now set correctly for all transports.

Contributed by Alex Koshelev.

- `qpid`: Fixed bug where the connection was not being closed properly.

Contributed by Brian Bouterse.

- `bindings` is now JSON serializable (Issue #453).

Contributed by Sergey Tikhonov.

- Fixed typo in error when `yaml` is not installed (said `msgpack`).

Contributed by Joshua Harlow.

- Redis: Now properly handles `redis.exceptions.TimeoutError` raised by `redis`.

Contributed by markow.

- `qpid`: Adds additional string to check for when connecting to `qpid`.

When we connect to `qpid`, we need to ensure that we skip to the next SASL mechanism if the current mechanism fails. Otherwise, we will keep retrying the connection with a non-working mech.

Contributed by Chris Duryee.

- `qpid`: Handle `NotFound` exceptions.

Contributed by Brian Bouterse.

- `Queue.__repr__` now makes sure return value is not unicode (Issue #440).

- `qpid`: `Queue.purge` incorrectly raised `AttributeError` if the does not exist (Issue #439).

Contributed by Brian Bouterse.

- Linux: Now ignores permission errors on `epoll` unregister.

## 5.14 3.0.24

**release-date** 2014-11-17 11:00 P.M UTC

**release-by** Ask Solem

- The `Qpid` broker is supported for Python 2.x environments. The `Qpid` transport includes full SSL support within Kombu. See the `kombu.transport.qpid` docs for more info.

Contributed by Brian Bouterse and Chris Duryee through support from Red Hat.

- Dependencies: `extra[librabbitmq]` now requires `librabbitmq 1.6.0`
- Docstrings for `TokenBucket` did not match implementation.

Fix contributed by Jesse Dhillon.

- `oid_from()` accidentally called `uuid.getnode()` but did not use the return value.

Fix contributed by Alexander Todorov.

- Redis: Now ignores errors when closing the underlying connection.
- Redis: Restoring messages will now use a single connection.
- `kombu.five.monotonic`: Can now be imported even if `ctypes` is not available for some reason (e.g. App Engine)
- Documentation: Improved example to use the `declare` argument to `Producer` (Issue #423).
- Django: Fixed `app_label` for older Django versions (`< 1.7`). (Issue #414).

## 5.15 3.0.23

**release-date** 2014-09-14 10:45 P.M UTC

**release-by** Ask Solem

- Django: Fixed bug in the Django 1.7 compatibility improvements related to autocommit handling.  
Contributed by Radek Czajka.
- Django: The Django transport models would not be created on syncdb after app label rename (Issue #406).

## 5.16 3.0.22

**release-date** 2014-09-04 03:00 P.M UTC

**release-by** Ask Solem

- `kombu.async`: Min. delay between waiting for timer was always increased to one second.
- Fixed bug in `itermessages` where message is received after the `with` statement exits the block.

Fixed by Romyana Neykova

- **Connection.autoretry: Now works with functions missing wrapped attributes** (`__module__`, `__name__`, `__doc__`). Fixes #392.

Contributed by johtso.

- Django: Now sets custom app label for `kombu.transport.django` to work with recent changes in Django 1.7.
- `SimpleQueue` removed messages from the wrong end of buffer (Issue #380).
- Tests: Now using `unittest.mock` if available (Issue #381).

## 5.17 3.0.21

**release-date** 2014-07-07 02:00 P.M UTC

**release-by** Ask Solem

- Fixed remaining bug in `maybe_declare` for `auto_delete` exchanges.  
Fix contributed by Roger Hu.
- MongoDB: Creating a channel now properly evaluates a connection (Issue #363).  
Fix contributed by Len Buckens.

## 5.18 3.0.20

**release-date** 2014-06-24 02:30 P.M UTC

**release-by** Ask Solem

- Reverts change in 3.0.17 where `maybe_declare` caches the declaration of `auto_delete` queues and exchanges.  
Fix contributed by Roger Hu.
- Redis: Fixed race condition when using `gevent` and the channel is closed.  
Fix contributed by Andrew Rodionoff.

## 5.19 3.0.19

**release-date** 2014-06-09 03:10 P.M UTC

**release-by** Ask Solem

- The wheel distribution did not support Python 2.6 by failing to list the extra dependencies required.
- Durable and `auto_delete` queues/exchanges can be cached using `maybe_declare`.

## 5.20 3.0.18

**release-date** 2014-06-02 06:00 P.M UTC

**release-by** Ask Solem

- A typo introduced in 3.0.17 caused `kombu.async.hub` to crash (Issue #360).

## 5.21 3.0.17

**release-date** 2014-06-02 05:00 P.M UTC

**release-by** Ask Solem

- `kombu[librabbitmq]` now depends on `librabbitmq` 1.5.2.
- Async: Event loop now selectively removes file descriptors for the mode it failed in, and keeps others (e.g read vs write).

Fix contributed by Roger Hu.

- CouchDB: Now works without `userid` set.

Fix contributed by Latitia M. Haskins.

- SQLAlchemy: Now supports recovery from connection errors.

Contributed by Felix Schwarz.

- Redis: Restore at shutdown now works when `ack_emulation` is disabled.
- `kombu.common.eventloop()` accidentally swallowed socket errors.
- Adds `kombu.utils.url.sanitize_url()`

## 5.22 3.0.16

**release-date** 2014-05-06 01:00 P.M UTC

**release-by** Ask Solem

- `kombu[librabbitmq]` now depends on `librabbitmq` 1.5.1.
- Redis: Fixes `TypeError` problem in `unregister` (Issue #342).

Fix contributed by Tobias Schottdorf.

- Tests: Some unit tests accidentally required the `redis-py` library.

Fix contributed by Randy Barlow.

- `librabbitmq`: Would crash when using an older version of `librabbitmq`, now emits warning instead.

## 5.23 3.0.15

**release-date** 2014-04-15 09:00 P.M UTC

**release-by** Ask Solem

- Now depends on `amqp` 1.4.5.
- RabbitMQ 3.3 changes QoS semantics (Issue #339).

See the RabbitMQ release notes here: <http://www.rabbitmq.com/blog/2014/04/02/breaking-things-with-rabbitmq-3-3/>

A new connection property has been added that can be used to detect whether the remote server is using this new QoS behavior:

```
>>> Connection('amqp://').qos_behavior_matches_spec
False
```

so if your application depends on the old semantics you can use this to set the `apply_global` flag appropriately:

```
def update_prefetch_count(channel, new_value):
    channel.basic_qos(
        0, new_value,
        not channel.connection.client.qos_behavior_matches_spec,
    )
```

- Users of `librabbitmq` is encouraged to upgrade to `librabbitmq 1.5.0`.

The `kombu[librabbitmq]` extra has been updated to depend on this version.

- Pools: Now takes transport options into account when comparing connections (Issue #333).
- MongoDB: Fixes Python 3 compatibility.
- Async: select: Ignore socket errors when attempting to unregister handles from the loop.
- Pidbox: Can now be configured to use a serializer other than `json`, but specifying a serializer argument to *Mailbox*.

Contributed by Dmitry Malinovsky.

- Message decompression now works with Python 3.

Fix contributed by Adam Gaca.

## 5.24 3.0.14

**release-date** 2014-03-19 07:00 P.M UTC

**release-by** Ask Solem

- **MongoDB**: Now endures a connection failover (Issue #123).

Fix contributed by Alex Koshelev.

- **MongoDB**: Fixed `KeyError` when a replica set member is removed.

Also fixes `celery#971` and `celery/#898`.

Fix contributed by Alex Koshelev.

- **MongoDB**: Fixed MongoDB broadcast cursor re-initialization bug.

Fix contributed by Alex Koshelev.

- **Async**: Fixed bug in `lax` semaphore implementation where in some usage patterns the limit was not honored correctly.

Fix contributed by Ionel Cristian Mărie.

- **Redis**: Fixed problem with fanout when using Python 3 (Issue #324).

- **Redis**: Fixed `AttributeError` from attempting to close a non-existing connection (Issue #320).

## 5.25 3.0.13

**release-date** 2014-03-03 04:00 P.M UTC

**release-by** Ask Solem

- **Redis**: Fixed serious race condition that could lead to data loss.

The delivery tags were accidentally set to be an incremental number local to the channel, but the delivery tags need to be globally unique so that a message can not overwrite an older message in the backup store.

This change is not backwards incompatible and you are encouraged to update all your system using a previous version as soon as possible.

- Now depends on `amqp` 1.4.4.
- `Pidbox`: Now makes sure message encoding errors are handled by default, so that a custom error handler does not need to be specified.
- `Redis`: The fanout exchange can now use AMQP patterns to route and filter messages.

This change is backwards incompatible and must be enabled with the `fanout_patterns` transport option:

```
>>> conn = kombu.Connection('redis://', transport_options={
...     'fanout_patterns': True,
... })
```

When enabled the exchange will work like an `amqp` topic exchange if the binding key is a pattern.

This is planned to be default behavior in the future.

- `Redis`: Fixed `cycle` no such attribute error.

## 5.26 3.0.12

**release-date** 2014-02-09 03:50 P.M UTC

**release-by** Ask Solem

- Now depends on `amqp` 1.4.3.
- Fixes Python 3.4 logging incompatibility (Issue #311).
- `Redis`: Now properly handles unknown pub/sub messages.

Fix contributed by Sam Stavinoha.

- `amqplib`: Fixed bug where more bytes were requested from the socket than necessary.

Fix contributed by Ionel Cristian Mărie.

## 5.27 3.0.11

**release-date** 2014-02-03 05:00 P.M UTC

**release-by** Ask Solem

- Now depends on `amqp` 1.4.2.
- Now always trusts messages of type `application/data` and `application/text` or which have an unspecified content type (Issue #306).
- Compression errors are now handled as decode errors and will trigger the `Consumer.on_decode_error` callback if specified.
- New `kombu.Connection.get_heartbeat_interval()` method that can be used to access the negotiated heartbeat value.
- ***kombu.common.oid\_for* no longer uses the MAC address of the host, but** instead uses a process-wide UUID4 as a node id.

This avoids a call to `uuid.getnode()` at module scope.

- `Hub.add`: Now normalizes registered `fileno`.

Contributed by Ionel Cristian Mărie.

- SQS: Fixed bug where the prefetch count limit was not respected.

## 5.28 3.0.10

**release-date** 2014-01-17 05:40 P.M UTC

**release-by** Ask Solem

- Now depends on `amqp 1.4.1`.
- `maybe_declare` now raises a “recoverable connection error” if the channel is disconnected instead of a `ChannelError` so that the operation can be retried.
- Redis: `Consumer.cancel()` is now thread safe.

This fixes an issue when using `gevent/eventlet` and a message is handled after the consumer is cancelled resulting in a “message for queue without consumers” error.

- Retry operations would not always respect the `interval_start` value when calculating the time to sleep for (Issue #303).

Fix contributed by Antoine Legrand.

- Timer: Fixed “unhashable type” error on Python 3.
- Hub: Do not attempt to unregister operations on an already closed poller instance.

## 5.29 3.0.9

**release-date** 2014-01-13 05:30 P.M UTC

**release-by** Ask Solem

- Now depends on `amqp 1.4.0`.
- Redis: Basic cancel for fanout based queues now sends a corresponding `UNSUBSCRIBE` command to the server.
- MongoDB: Improved connection string and options handling (Issue #266 + Issue #120).

Contributed by Alex Koshelev.

- SQS: Limit the number of messages when receiving in batch to 10.

This is a hard limit enforced by Amazon so the `sq3` transport must not exceed this value.

Fix contributed by Eric Reynolds.

- ConsumerMixin: `consume` now checks heartbeat every time the socket times out.

Contributed by Dustin J. Mitchell.

- Retry Policy: A max retries of 0 did not retry forever.

Fix contributed by Antoine Legrand.

- Simple: If passing a Queue object the simple utils will now take default routing key from that queue.

Contributed by Fernando Jorge Mota.

- `repr` (producer) no longer evaluates the underlying channel.
- Redis: The map of Redis error classes are now exposed at the module level using the `kombu.transport.redis.get_redis_error_classes()` function.
- Async: `Hub.close` now sets `.poller` to `None`.

## 5.30 3.0.8

**release-date** 2013-12-16 05:00 P.M UTC

**release-by** Ask Solem

- Serializer: `loads` and `dumps` now wraps exceptions raised into `DecodeError` and `kombu.exceptions.EncodeError` respectively.

Contributed by Ionel Cristian Maries

- Redis: Would attempt to read from the wrong connection if a `select/epoll/kqueue` exception event happened.

Fix contributed by Michael Nelson.

- Redis: Disabling `ack emulation` now works properly.

Fix contributed by Michael Nelson.

- Redis: `IOError` and `OSError` are now treated as recoverable connection errors.

- SQS: Improved performance by reading messages in bulk.

Contributed by Matt Wise.

- Connection Pool: Attempting to acquire from a closed pool will now raise `RuntimeError`.

## 5.31 3.0.7

**release-date** 2013-12-02 04:00 P.M UTC

**release-by** Ask Solem

- Fixes Python 2.6 compatibility.
- Redis: Fixes 'bad file descriptor' issue.

## 5.32 3.0.6

**release-date** 2013-11-21 04:50 P.M UTC

**release-by** Ask Solem

- Timer: No longer attempts to hash keyword arguments (Issue #275).
- Async: Did not account for the long type for file descriptors.

Fix contributed by Fabrice Rabaute.

- PyPy: `kqueue` support was broken.
- Redis: Bad `pub/sub` payloads no longer crashes the consumer.



- Redis: Unix socket URLs can now specify a virtual host by including it as a query parameter.

Example URL specifying a virtual host using database number 3:

```
redis+socket:///tmp/redis.sock?virtual_host=3
```

- `kombu.VERSION` is now a named tuple.

## 5.33 3.0.5

**release-date** 2013-11-15 11:00 P.M UTC

**release-by** Ask Solem

- Now depends on `amqp` 1.3.3.
- Redis: Fixed Python 3 compatibility problem (Issue #270).
- MongoDB: Fixed problem with URL parsing when authentication used.  
Fix contributed by dongweiming.
- `pyamqp`: Fixed small issue when publishing the message and the property dictionary was set to `None`.  
Fix contributed by Victor Garcia.
- Fixed problem in `repr(LaxBoundedSemaphore)`.  
Fix contributed by Antoine Legrand.
- Tests now passing on Python 3.3.

## 5.34 3.0.4

**release-date** 2013-11-08 01:00 P.M UTC

**release-by** Ask Solem

- `common.QoS.decrement_eventually` now makes sure the value does not go below 1 if a prefetch count is enabled.

## 5.35 3.0.3

**release-date** 2013-11-04 03:00 P.M UTC

**release-by** Ask Solem

- SQS: Properly reverted patch that caused delays between messages.  
Contributed by James Saryerwinnie
- `select`: Clear all registered fds on `poller.close`
- `Eventloop`: unregister if `EBADF` raised.

## 5.36 3.0.2

**release-date** 2013-10-29 02:00 P.M UTC

**release-by** Ask Solem

- Now depends on amqp version 1.3.2.
- select: Fixed problem where unregister did not properly remove the fd.

## 5.37 3.0.1

**release-date** 2013-10-24 04:00 P.M UTC

**release-by** Ask Solem

- Now depends on amqp version 1.3.1.
- Redis: New option `fanout_keyprefix`

This transport option is recommended for all users as it ensures that broadcast (fanout) messages sent is only seen by the current virtual host:

```
Connection('redis://', transport_options={'fanout_keyprefix': True})
```

However, enabling this means that you cannot send or receive messages from older Kombu versions so make sure all of your participants are upgraded and have the transport option enabled.

This will be the default behavior in Kombu 4.0.

- Distribution: Removed file `requirements/py25.txt`.
- MongoDB: Now disables `auto_start_request`.
- MongoDB: Enables `use_greenlets` if `eventlet/gevent` used.
- Pidbox: Fixes problem where expires header was None, which is a value not supported by the amq protocol.
- ConsumerMixin: New `consumer_context` method for starting the consumer without draining events.

## 5.38 3.0.0

**release-date** 2013-10-14 04:00 P.M BST

**release-by** Ask Solem

- Now depends on amqp version 1.3.
- No longer supports Python 2.5

The minimum Python version supported is now Python 2.6.0 for Python2, and Python 3.3 for Python3.

- Dual codebase supporting both Python 2 and 3.

No longer using `2to3`, making it easier to maintain support for both versions.

- pickle, yaml and msgpack deserialization is now disabled by default.

This means that Kombu will by default refuse to handle any content type other than json.

Pickle is known to be a security concern as it will happily load any object that is embedded in a pickle payload, and payloads can be crafted to do almost anything you want. The default serializer in Kombu is json but it also supports a number of other serialization formats that it will evaluate if received: including pickle.

It was always assumed that users were educated about the security implications of pickle, but in hindsight we don't think users should be expected to secure their services if we have the ability to be secure by default.

By disabling any content type that the user did not explicitly want enabled we ensure that the user must be conscious when they add pickle as a serialization format to support.

The other built-in serializers (yaml and msgpack) are also disabled even though they aren't considered insecure <sup>1</sup> at this point. Instead they're disabled so that if a security flaw is found in one of these libraries in the future, you will only be affected if you have explicitly enabled them.

To have your consumer accept formats other than json you have to explicitly add the wanted formats to a white-list of accepted content types:

```
>>> c = Consumer(conn, accept=['json', 'pickle', 'msgpack'])
```

or when using synchronous access:

```
>>> msg = queue.get(accept=['json', 'pickle', 'msgpack'])
```

The `accept` argument was first supported for consumers in version 2.5.10, and first supported by `Queue.get` in version 2.5.15 so to stay compatible with previous versions you can enable the previous behavior:

```
>>> from kombu import enable_insecure_serializers
>>> enable_insecure_serializers()
```

But note that this has global effect, so be very careful should you use it.

- `kombu.async`: Experimental event loop implementation.

This code was previously in Celery but was moved here to make it easier for async transport implementations.

The API is meant to match the Tulip API which will be included in Python 3.4 as the `asyncio` module. It's not a complete implementation obviously, but the goal is that it will be easy to change to it once that is possible.

- Utility function `kombu.common.ipublish` has been removed.

Use `Producer(..., retry=True)` instead.

- Utility function `kombu.common.isend_reply` has been removed

Use `send_reply(..., retry=True)` instead.

- `kombu.common.entry_to_queue` and `kombu.messaging.entry_to_queue` has been removed.

Use `Queue.from_dict(name, **options)` instead.

- Redis: Messages are now restored at the end of the list.

Contributed by Mark Lavin.

<sup>1</sup> The PyYAML library has a `yaml.load()` function with some of the same security implications as pickle, but Kombu uses the `yaml.safe_load()` function which is not known to be affected.

- **StdConnectionError and StdChannelError is removed** and `amqp.ConnectionError` and `amqp.ChannelError` is used instead.
- Message object implementation has moved to `kombu.message.Message`.
- Serailization: Renamed functions `encode/decode` to `dumps ()` and `loads ()`.  
For backward compatibility the old names are still available as aliases.
- The `kombu.log.anon_logger` function has been removed.  
Use `get_logger ()` instead.
- `queue_declare` now returns namedtuple with `queue`, `message_count`, and `consumer_count` fields.
- LamportClock: Can now set lock class
- `kombu.utils.clock`: Utilities for ordering events added.
- `SimpleQueue` now allows you to override the exchange type used.  
Contributed by Vince Gonzales.
- Zookeeper transport updated to support new changes in the `kazoo` library.  
Contributed by Mahendra M.
- **pyamqp/librabbitmq: Transport options are now forwarded as keyword arguments** to the underlying connection (Issue #214).
- Transports may now distinguish between recoverable and irrecoverable connection and channel errors.
- `kombu.utils.Finalize` has been removed: Use `multiprocessing.util.Finalize` instead.
- Memory transport now supports the fanout exchange type.  
Contributed by Davanum Srinivas.
- Experimental new `Pyro` transport (`kombu.transport.pyro`).  
Contributed by Tommie McAfee.
- Experimental new `SoftLayer MQ` transport (`kombu.transport.SLMQ`).  
Contributed by Kevin McDonald
- Eventio: Kqueue breaks in subtle ways so `select` is now used instead.
- SQLAlchemy transport: Can now specify table names using the `queue_tablename` and `message_tablename` transport options.  
Contributed by Ryan Petrello.

**Redis transport: Now supports using local UNIX sockets to communicate with the** Redis server (Issue #1283)

To connect using a UNIX socket you have to use the `redis+socket` URL-prefix:  
`redis+socket:///tmp/redis.sock`.

This functionality was merged from the `celery-redis-unixsocket` project. Contributed by Maxime Rouyre.

ZeroMQ transport: `drain_events` now supports timeout.

Contributed by Jesper Thomschütz.

## 5.39 2.5.16

**release-date** 2013-10-04 03:30 P.M BST

**release-by** Ask Solem

- Python3: Fixed problem with dependencies not being installed.

## 5.40 2.5.15

**release-date** 2013-10-04 03:30 P.M BST

**release-by** Ask Solem

- Declaration cache: Now only keeps hash of declaration so that it does not keep a reference to the channel.
- Declaration cache: Now respects `entity.can_cache_declaration` attribute.
- Fixes Python 2.5 compatibility.
- Fixes tests after python-msgpack changes.
- `Queue.get`: Now supports `accept` argument.

## 5.41 2.5.14

**release-date** 2013-08-23 05:00 P.M BST

**release-by** Ask Solem

- `safe_str` did not work properly resulting in `UnicodeDecodeError` (Issue #248).

## 5.42 2.5.13

**release-date** 2013-08-16 04:00 P.M BST

**release-by** Ask Solem

- Now depends on `amqp` 1.0.13
- Fixed typo in Django functional tests.
- `safe_str` now returns Unicode in Python 2.x
  - Fix contributed by Germán M. Bravo.
- `amqp`: Transport options are now merged with arguments supplied to the connection.
- Tests no longer depends on `distribute`, which was deprecated and merged back into `setuptools`.
  - Fix contributed by Sascha Peilicke.
- `ConsumerMixin` now also restarts on channel related errors.
  - Fix contributed by Corentin Ardeois.

## 5.43 2.5.12

**release-date** 2013-06-28 03:30 P.M BST

**release-by** Ask Solem

- Redis: Ignore errors about keys missing in the round-robin cycle.
- Fixed test suite errors on Python 3.
- Fixed msgpack test failures.

## 5.44 2.5.11

**release-date** 2013-06-25 02:30 P.M BST

**release-by** Ask Solem

- Now depends on amqp 1.0.12 (Py3 compatibility issues).
- MongoDB: Removed cause of a “database name in URI is being ignored” warning.  
Fix by Flavio Percoco Premoli
- Adds `passive` option to `Exchange`.  
Setting this flag means that the exchange will not be declared by kombu, but that it must exist already (or an exception will be raised).  
Contributed by Rafal Malinowski
- `Connection.info()` now gives the current hostname and not the list of available hostnames.  
Fix contributed by John Shuping.
- `pyamqp`: Transport options are now forwarded as kwargs to `amqp.Connection`.
- `librabbitmq`: Transport options are now forwarded as kwargs to `librabbitmq.Connection`.
- `librabbitmq`: Now raises `NotImplementedError` if SSL is enabled.  
The `librabbitmq` library does not support `ssl`, but you can use `stunnel` or change to the `pyamqp://transport` instead.  
Fix contributed by Dan LaMotte.
- `librabbitmq`: Fixed a cyclic reference at connection close.
- `eventio`: select implementation now removes bad file descriptors.
- `eventio`: Fixed Py3 compatibility problems.
- Functional tests added for `py-amqp` and `librabbitmq` transports.
- `Resource.force_close_all` no longer uses a mutex.
- `Pidbox`: Now ignores `InconsistencyError` when sending replies, as this error simply means that the client may no longer be alive.
- Adds new `Connection.collect` method, that can be used to clean up after connections without I/O.
- `queue_bind` is no longer called for queues bound to the “default exchange” (Issue #209).  
Contributed by Jonathan Halcrow.
- The `max_retries` setting for retries was not respected correctly (off by one).

## 5.45 2.5.10

**release-date** 2013-04-11 06:10 P.M BST

**release-by** Ask Solem

### 5.45.1 Note about upcoming changes for Kombu 3.0

Kombu 3 consumers will no longer accept pickle/yaml or msgpack by default, and you will have to explicitly enable untrusted deserializers either globally using `kombu.enable_insecure_serializers()`, or using the `accept` argument to `Consumer`.

### 5.45.2 Changes

- New utility function to disable/enable untrusted serializers.
  - `kombu.disable_insecure_serializers()`
  - `kombu.enable_insecure_serializers()`.
- Consumer: `accept` can now be used to specify a whitelist of content types to accept.

If the `accept` whitelist is set and a message is received with a content type that is not in the whitelist then a `ContentDisallowed` exception is raised. Note that this error can be handled by the already existing `on_decode_error` callback

Examples:

```
Consumer(accept=['application/json'])
Consumer(accept=['pickle', 'json'])
```

- Now depends on amqp 1.0.11
- pidbox: Mailbox now supports the `accept` argument.
- Redis: More friendly error for when keys are missing.
- Connection URLs: The parser did not work well when there were multiple '+' tokens.

## 5.46 2.5.9

**release-date** 2013-04-08 05:07 P.M BST

**release-by** Ask Solem

- Pidbox: Now warns if there are multiple nodes consuming from the same pidbox.
- Adds `Queue.on_declared`
  - A callback to be called when the queue is declared, with signature `(name, messages, consumers)`.
- Now uses fuzzy matching to suggest alternatives to typos in transport names.
- SQS: Adds new transport option `queue_prefix`.
  - Contributed by j0hnsmith.
- pyamqp: No longer overrides `verify_connection`.

- SQS: Now specifies the `driver_type` and `driver_name` attributes.  
Fix contributed by Mher Movsisyan.
- Fixed bug with `kombu.utils.retry_over_time` when no errback specified.

## 5.47 2.5.8

**release-date** 2013-03-21 04:00 P.M UTC

**release-by** Ask Solem

- Now depends on `amqp` 1.0.10 which fixes a Python 3 compatibility error.
- Redis: Fixed a possible race condition (Issue #171).
- Redis: Ack emulation/`visibility_timeout` can now be disabled using a transport option.

Ack emulation adds quite a lot of overhead to ensure data is safe even in the event of an unclean shutdown. If data loss do not worry you there is now an `ack_emulation` transport option you can use to disable it:

```
Connection('redis://', transport_options={'ack_emulation': False})
```

- SQS: Fixed `botocore` v2.7 compatibility (Issue #207).
- Exchange: Should not try to re-declare default exchange ("") (Issue #209).
- SQS: Long polling is now disabled by default as it was not implemented correctly, resulting in long delays between receiving messages (Issue #202).
- Fixed Python 2.6 incompatibility depending on `exc.errno` being available.  
Fix contributed by Ephemera.

## 5.48 2.5.7

**release-date** 2013-03-08 01:00 P.M UTC

**release-by** Ask Solem

- Now depends on `amqp` 1.0.9
- Redis: A regression in 2.5.6 caused the redis transport to ignore options set in `transport_options`.
- Redis: New `socket_timeout` transport option.
- Redis: `InconsistencyError` is now regarded as a recoverable error.
- Resource pools: Will no longer attempt to release resource that was never acquired.
- MongoDB: Now supports the `ssl` option.

Contributed by Sebastian Pawlus.



## 5.49 2.5.6

**release-date** 2013-02-08 01:00 P.M UTC

**release-by** Ask Solem

- Now depends on amqp 1.0.8 which works around a bug found on some Python 2.5 installations where `2**32` overflows to 0.

## 5.50 2.5.5

**release-date** 2013-02-07 05:00 P.M UTC

**release-by** Ask Solem

SQS: Now supports long polling (Issue #176).

The polling interval default has been changed to 0 and a new transport option (`wait_time_seconds`) has been added. This parameter specifies how long to wait for a message from SQS, and defaults to 20 seconds, which is the maximum value currently allowed by Amazon SQS.

Contributed by James Saryerwinnie.

- SQS: Now removes unpickleable fields before restoring messages.
- `Consumer.__exit__` now ignores exceptions occurring while cancelling the consumer.
- Virtual: Routing keys can now consist of characters also used in regular expressions (e.g. parens) (Issue #194).
- Virtual: Fixed compression header when restoring messages.

Fix contributed by Alex Koshelev.

- Virtual: `ack/reject/requeue` now works while using `basic_get`.
- Virtual: `Message.reject` is now supported by virtual transports (`requeue` depends on individual transport support).
- Fixed typo in hack used for static analyzers.

Fix contributed by Basil Mironenko.

## 5.51 2.5.4

**release-date** 2012-12-10 12:35 P.M UTC

**release-by** Ask Solem

- Fixed problem with connection clone and multiple URLs (Issue #182).

Fix contributed by Dane Guempel.

- `zeromq`: Now compatible with `libzmq 3.2.x`.

Fix contributed by Andrey Antukh.

- Fixed Python 3 installation problem (Issue #187).

## 5.52 2.5.3

**release-date** 2012-11-29 12:35 P.M UTC

**release-by** Ask Solem

- Pidbox: Fixed compatibility with Python 2.6

## 5.53 2.5.2

**release-date** 2012-11-29 12:35 P.M UTC

**release-by** Ask Solem

## 5.54 2.5.2

**release-date** 2012-11-29 12:35 P.M UTC

**release-by** Ask Solem

- [Redis] Fixed connection leak and added a new 'max\_connections' transport option.

## 5.55 2.5.1

**release-date** 2012-11-28 12:45 P.M UTC

**release-by** Ask Solem

- Fixed bug where return value of `Queue.as_dict` could not be serialized with JSON (Issue #177).

## 5.56 2.5.0

**release-date** 2012-11-27 04:00 P.M UTC

**release-by** Ask Solem

- `py-amqp` is now the new default transport, replacing `amqplib`.

The new `py-amqp` library is a fork of `amqplib` started with the following goals:

- Uses AMQP 0.9.1 instead of 0.8
- Support for heartbeats (Issue #79 + Issue #131)
- Automatically revives channels on channel errors.
- **Support for all RabbitMQ extensions**
  - \* Consumer Cancel Notifications (Issue #131)
  - \* Publisher Confirms (Issue #131).
  - \* Exchange-to-exchange bindings: `exchange_bind` / `exchange_unbind`.

- API compatible with `librabbitmq` so that it can be used as a pure-python replacement in environments where `rabbitmq-c` cannot be compiled. `librabbitmq` will be updated to support all the same features as `py-amqp`.
- Support for using multiple connection URL's for failover.

The first argument to `Connection` can now be a list of connection URLs:

```
Connection(['amqp://foo', 'amqp://bar'])
```

or it can be a single string argument with several URLs separated by semicolon:

```
Connection('amqp://foo;amqp://bar')
```

There is also a new keyword argument `failover_strategy` that defines how `ensure_connection()`/`ensure()/kombu.Connection.autoretry()` will reconnect in the event of connection failures.

The default reconnection strategy is `round-robin`, which will simply cycle through the list forever, and there's also a `shuffle` strategy that will select random hosts from the list. Custom strategies can also be used, in that case the argument must be a generator yielding the URL to connect to.

Example:

```
Connection('amqp://foo;amqp://bar')
```

- Now supports PyDev, PyCharm, pylint and other static code analysis tools.
- `Queue` now supports multiple bindings.

You can now have multiple bindings in the same queue by having the second argument be a list:

```
from kombu import binding, Queue

Queue('name', [
    binding(Exchange('E1'), routing_key='foo'),
    binding(Exchange('E1'), routing_key='bar'),
    binding(Exchange('E2'), routing_key='baz'),
])
```

To enable this, helper methods have been added:

- `bind_to()`
- `unbind_from()`

Contributed by Rumyana Neykova.

- Custom serializers can now be registered using `Setuptools` entry-points. See *Creating extensions using `Setuptools` entry-points*.
- New `kombu.common.QoS` class used as a thread-safe way to manage changes to a consumer or channels `prefetch_count`. This was previously an internal class used in Celery now moved to the `kombu.common` module.
- Consumer now supports a `on_message` callback that can be used to process raw messages (not decoded). Other callbacks specified using the `callbacks` argument, and the `receive` method will be not be called when a `on_message` callback is present.
- New utility `kombu.common.ignore_errors()` ignores connection and channel errors.

Must only be used for cleanup actions at shutdown or on connection loss.

- Support for exchange-to-exchange bindings.

The *Exchange* entity gained `bind_to` and `unbind_from` methods:

```
e1 = Exchange('A')(connection)
e2 = Exchange('B')(connection)

e2.bind_to(e1, routing_key='rkey', arguments=None)
e2.unbind_from(e1, routing_key='rkey', arguments=None)
```

This is currently only supported by the `pyamqp` transport.

Contributed by Romyana Neykova.

## 5.57 2.4.10

**release-date** 2012-11-22 06:00 P.M UTC

**release-by** Ask Solem

- The previous versions connection pool changes broke Redis support so that it would always connect to localhost (default setting) no matter what connection parameters were provided (Issue #176).

## 5.58 2.4.9

**release-date** 2012-11-21 03:00 P.M UTC

**release-by** Ask Solem

- Redis: Fixed race condition that could occur while trying to restore messages (Issue #171).  
Fix contributed by Ollie Walsh.
- Redis: Each channel is now using a specific connection pool instance, which is disconnected on connection failure.
- ProducerPool: Fixed possible dead-lock in the `acquire` method.
- ProducerPool: `force_close_all` no longer tries to call the non-existent `Producer._close`.
- `librabbitmq`: Now implements `transport.verify_connection` so that connection pools will not give back connections that are no longer working.
- New and better `repr()` for Queue and Exchange objects.
- Python3: Fixed problem with running the unit test suite.
- Python3: Fixed problem with JSON codec.

## 5.59 2.4.8

**release-date** 2012-11-02 05:00 P.M UTC

**release-by** Ask Solem

- Redis: Improved fair queue cycle implementation (Issue #166).

Contributed by Kevin McCarthy.

- Redis: Unacked message restore limit is now unlimited by default.

Also, the limit can now be configured using the `unacked_restore_limit` transport option:

```
Connection('redis://', transport_options={
    'unacked_restore_limit': 100,
})
```

A limit of `100` means that the consumer will restore at most `100` messages at each `pass`.

- Redis: Now uses a mutex to ensure only one consumer restores messages at a time.

The mutex expires after 5 minutes by default, but can be configured using the `unacked_mutex_expire` transport option.

- `LamportClock.adjust` now returns the new clock value.
- Heartbeats can now be specified in URLs.

Fix contributed by Mher Movsisyan.

- Kombu can now be used with PyDev, PyCharm and other static analysis tools.
- Fixes problem with msgpack on Python 3 (Issue #162).

Fix contributed by Jasper Bryant-Greene

- `amqpplib`: Fixed bug with timeouts when SSL is used in non-blocking mode.

Fix contributed by Mher Movsisyan

## 5.60 2.4.7

**release-date** 2012-09-18 03:00 P.M BST

**release-by** Ask Solem

- Virtual: Unknown exchanges now default to 'direct' when sending a message.
- MongoDB: Fixed memory leak when merging keys stored in the db (Issue #159)

Fix contributed by Michael Korbakov.

- MongoDB: Better index for MongoDB transport (Issue #158).

This improvement will create a new compound index for queue and `_id` in order to be able to use both indexed fields for getting a new message (using queue field) and sorting by `_id`. It'll be necessary to manually delete the old index from the collection.

Improvement contributed by rmihael

## 5.61 2.4.6

**release-date** 2012-09-12 03:00 P.M BST

**release-by** Ask Solem

- Adds additional compatibility dependencies:

- Python <= 2.6:
  - \* importlib
  - \* ordereddict
- Python <= 2.5
  - \* simplejson

## 5.62 2.4.5

**release-date** 2012-08-30 03:36 P.M BST

**release-by** Ask Solem

- Last version broke installation on PyPy and Jython due to test requirements clean-up.

## 5.63 2.4.4

**release-date** 2012-08-29 04:00 P.M BST

**release-by** Ask Solem

- amqpplib: Fixed a bug with asynchronously reading large messages.
- pyamqp: Now requires amqp 0.9.3
- Cleaned up test requirements.

## 5.64 2.4.3

**release-date** 2012-08-25 10:30 P.M BST

**release-by** Ask Solem

- Fixed problem with amqp transport alias (Issue #154).

## 5.65 2.4.2

**release-date** 2012-08-24 05:00 P.M BST

**release-by** Ask Solem

- Having an empty transport name broke in 2.4.1.

## 5.66 2.4.1

**release-date** 2012-08-24 04:00 P.M BST

**release-by** Ask Solem

- Redis: Fixed race condition that could cause the consumer to crash (Issue #151)

Often leading to the error message "could not convert string to float"

- Connection retry could cause an infinite loop (Issue #145).
- The `amqp` alias is now resolved at runtime, so that eventlet detection works even if patching was done later.

## 5.67 2.4.0

**release-date** 2012-08-17 08:00 P.M BST

**release-by** Ask Solem

- New experimental ZeroMQ `<kombu.transport.zmq transport`.  
Contributed by John Watson.
- Redis: Ack timed-out messages were not restored when using the eventloop.
- Now uses pickle protocol 2 by default to be cross-compatible with Python 3.  
The protocol can also now be changed using the `PICKLE_PROTOCOL` environment variable.
- Adds `Transport.supports_ev` attribute.
- Pika: Queue purge was not working properly.  
Fix contributed by Steeve Morin.
- Pika backend was no longer working since Kombu 2.3  
Fix contributed by Steeve Morin.

## 5.68 2.3.2

**release-date** 2012-08-01 06:00 P.M BST

**release-by** Ask Solem

- Fixes problem with deserialization in Python 3.

## 5.69 2.3.1

**release-date** 2012-08-01 04:00 P.M BST

**release-by** Ask Solem

- `librabbitmq`: Can now handle messages that does not have a `content_encoding/content_type` set (Issue #149).  
Fix contributed by C Anthony Risinger.
- `Beanstalk`: Now uses `localhost` by default if the URL does not contain a host.

## 5.70 2.3.0

**release-date** 2012-07-24 03:50 P.M BST

**release-by** Ask Solem

- New `pyamqp://` transport!

The new `py-amqp` library is a fork of `amqplib` started with the following goals:

- Uses AMQP 0.9.1 instead of 0.8
- Should support all RabbitMQ extensions
- API compatible with `librabbitmq` so that it can be used as a pure-python replacement in environments where `rabbitmq-c` cannot be compiled.

If you start using `py-amqp` instead of `amqplib` you can enjoy many advantages including:

- Heartbeat support (Issue #79 + Issue #131)
- Consumer Cancel Notifications (Issue #131)
- Publisher Confirms

`amqplib` has not been updated in a long while, so maintaining our own fork ensures that we can quickly roll out new features and fixes without resorting to monkey patching.

To use the `py-amqp` transport you must install the `amqp` library:

```
$ pip install amqp
```

and change the connection URL to use the correct transport:

```
>>> conn = Connection('pyamqp://guest:guest@localhost//')
```

The `pyamqp://` transport will be the default fallback transport in Kombu version 3.0, when `librabbitmq` is not installed, and `librabbitmq` will also be updated to support the same features.

- Connection now supports heartbeat argument.

If enabled you must make sure to manually maintain heartbeats by calling the `Connection.heartbeat_check` at twice the rate of the specified heartbeat interval.

E.g. if you have `Connection(heartbeat=10)`, then you must call `Connection.heartbeat_check()` every 5 seconds.

if the server has not sent heartbeats at a suitable rate then the heartbeat check method must raise an error that is listed in `Connection.connection_errors`.

The attribute `Connection.supports_heartbeats` has been added for the ability to inspect if a transport supports heartbeats or not.

Calling `heartbeat_check` on a transport that does not support heartbeats results in a noop operation.

- SQS: Fixed bug with invalid characters in queue names.

Fix contributed by Zach Smith.

- `utils.reprcall`: Fixed typo where `kwargs` argument was an empty tuple by default, and not an empty dict.

## 5.71 2.2.6

**release-date** 2012-07-10 05:00 P.M BST

**release-by** Ask Solem

- Adds `kombu.messaging.entry_to_queue` for compat with previous versions.



## 5.72 2.2.5

**release-date** 2012-07-10 05:00 P.M BST

**release-by** Ask Solem

- Pidbox: Now sets queue expire at 10 seconds for reply queues.
- EventIO: Now ignores `ValueError` raised by `epoll.unregister`.
- MongoDB: Fixes Issue #142

Fix by Flavio Percoco Premoli

## 5.73 2.2.4

**release-date** 2012-07-05 04:00 P.M BST

**release-by** Ask Solem

- Support for `msgpack-python 0.2.0` (Issue #143)

The latest `msgpack` version no longer supports Python 2.5, so if you're still using that you need to depend on an earlier `msgpack-python` version.

Fix contributed by Sebastian Insua

- `maybe_declare()` no longer caches entities with the `auto_delete` flag set.
- New experimental filesystem transport.  
Contributed by Bobby Beever.
- Virtual Transports: Now support anonymous queues and exchanges.

## 5.74 2.2.3

**release-date** 2012-06-24 05:00 P.M BST

**release-by** Ask Solem

- `BrokerConnection` now renamed to `Connection`.

The name `Connection` has been an alias for a very long time, but now the rename is official in the documentation as well.

The `Connection` alias has been available since version 1.1.3, and `BrokerConnection` will still work and is not deprecated.

- `Connection.clone()` now works for the `sqlalchemy` transport.
- `kombu.common.eventloop()`, `kombu.utils.uuid()`, and `kombu.utils.url.parse_url()` can now be imported from the `kombu` module directly.
- Pidbox transport callback `after_reply_message_received` now happens in a finally block.
- Trying to use the `librabbitmq://` transport will now show the right name in the `ImportError` if `librabbitmq` is not installed.

The `librabbitmq` falls back to the older `pylibrabbitmq` name for compatibility reasons and would therefore show `No module named pylibrabbitmq` instead of `librabbitmq`.

## 5.75 2.2.2

**release-date** 2012-06-22 02:30 P.M BST

**release-by** Ask Solem

- Now depends on `anyjson` 0.3.3
- Json serializer: Now passes `buffer` objects directly, since this is supported in the latest `anyjson` version.
- Fixes blocking `epoll` call if `timeout` was set to 0.

Fix contributed by John Watson.

- `setup.py` now takes requirements from the `requirements/` directory.
- The distribution directory `contrib/` is now renamed to `extra/`

## 5.76 2.2.1

**release-date** 2012-06-21 01:00 P.M BST

**release-by** Ask Solem

- SQS: Default visibility timeout is now 30 minutes.

Since we have ack emulation the visibility timeout is only in effect if the consumer is abruptly terminated.

- `retry` argument to `Producer.publish` now works properly, when the `declare` argument is specified.
- Json serializer: didn't handle `buffer` objects (Issue #135).

Fix contributed by Jens Hoffrichter.

- Virtual: Now supports `passive` argument to `exchange_declare`.
- Exchange & Queue can now be bound to connections (which will use the default channel):

```
>>> exchange = Exchange('name')
>>> bound_exchange = exchange(connection)
>>> bound_exchange.declare()
```

- `SimpleQueue` & `SimpleBuffer` can now be bound to connections (which will use the default channel).
- `Connection.manager.get_bindings` now works for `librabbitmq` and `pika`.
- Adds new transport info attributes:

```
- ``Transport.driver_type``
    Type of underlying driver, e.g. "amqp", "redis", "sql".

- ``Transport.driver_name``
    Name of library used e.g. "amqplib", "redis", "pymongo".

- ``Transport.driver_version()``
    Version of underlying library.
```

## 5.77 2.2.0

**release-date** 2012-06-07 03:10 P.M BST

**release-by** Ask Solem

### 5.77.1 Important Notes

- The canonical source code repository has been moved to

<http://github.com/celery/kombu>

- Pidbox: Exchanges used by pidbox are no longer auto\_delete.

Auto delete has been described as a misfeature, and therefore we have disabled it.

For RabbitMQ users old exchanges used by pidbox must be removed, these are named `mailbox_name.pidbox`, and `reply.mailbox_name.pidbox`.

The following command can be used to clean up these exchanges:

```
VHOST=/ URL=amqp:// python -c'import sys,kombu;[kombu.Connection(
    sys.argv[-1]).channel().exchange_delete(x)
    for x in sys.argv[1:-1]]' \
$(sudo rabbitmqctl -q list_exchanges -p "$VHOST" \
| grep \.pidbox | awk '{print $1}') "$URL"
```

The `VHOST` variable must be set to the target RabbitMQ virtual host, and the `URL` must be the AMQP URL to the server.

- The `amqp` transport alias will now use `librabbitmq` if installed.

`py-librabbitmq` is a fast AMQP client for Python using the `librabbitmq` C library.

It can be installed by:

```
$ pip install librabbitmq
```

It will not be used if the process is monkey patched by `eventlet/gevent`.

### 5.77.2 News

- Redis: Ack emulation improvements.

Reducing the possibility of data loss.

Acks are now implemented by storing a copy of the message when the message is consumed. The copy is not removed until the consumer acknowledges or rejects it.

This means that unacknowledged messages will be redelivered either when the connection is closed, or when the visibility timeout is exceeded.

- Visibility timeout

This is a timeout for acks, so that if the consumer does not ack the message within this time limit, the message is redelivered to another consumer.

The timeout is set to one hour by default, but can be changed by configuring a transport option:

```
>>> Connection('redis://', transport_options={
...     'visibility_timeout': 1800, # 30 minutes
... })
```

**NOTE:** Messages that have not been acked will be redelivered if the visibility timeout is exceeded, for Celery users this means that ETA/countdown tasks that are scheduled to execute with a time that exceeds the visibility timeout will be executed twice (or more). If you plan on using long ETA/countdowns you should tweak the visibility timeout accordingly:

```
BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 18000} # 5 hours
```

Setting a long timeout means that it will take a long time for messages to be redelivered in the event of a power failure, but if so happens you could temporarily set the visibility timeout lower to flush out messages when you start up the systems again.

- Experimental [Apache ZooKeeper](#) transport

More information is in the module reference: `kombu.transport.zookeeper`.

Contributed by Mahendra M.

- Redis: Priority support.

The message's `priority` field is now respected by the Redis transport by having multiple lists for each named queue. The queues are then consumed by in order of priority.

The priority field is a number in the range of 0 - 9, where 0 is the default and highest priority.

The priority range is collapsed into four steps by default, since it is unlikely that nine steps will yield more benefit than using four steps. The number of steps can be configured by setting the `priority_steps` transport option, which must be a list of numbers in **sorted order**:

```
>>> x = Connection('redis://', transport_options={
...     'priority_steps': [0, 2, 4, 6, 8, 9],
... })
```

Priorities implemented in this way is not as reliable as priorities on the server side, which is why nickname the feature “quasi-priorities”; **Using routing is still the suggested way of ensuring quality of service**, as client implemented priorities fall short in a number of ways, e.g. if the worker is busy with long running tasks, has prefetched many messages, or the queues are congested.

Still, it is possible that using priorities in combination with routing can be more beneficial than using routing or priorities alone. Experimentation and monitoring should be used to prove this.

Contributed by Germán M. Bravo.

- Redis: Now cycles queues so that consuming is fair.

This ensures that a very busy queue won't block messages from other queues, and ensures that all queues have an equal chance of being consumed from.

This used to be the case before, but the behavior was accidentally changed while switching to using blocking pop.

- Redis: Auto delete queues that are bound to fanout exchanges is now deleted at `channel.close`.
- `amqplib`: Refactored the `drain_events` implementation.
- `Pidbox`: Now uses `connection.default_channel`.
- Pickle serialization: Can now decode buffer objects.
- Exchange/Queue declarations can now be cached even if the entity is non-durable.

This is possible because the list of cached declarations are now kept with the connection, so that the entities will be redeclared if the connection is lost.

- Kombu source code now only uses one-level of explicit relative imports.

### 5.77.3 Fixes

- `eventio`: Now ignores `ENOENT` raised by `epoll.register`, and `EEXIST` from `epoll.unregister`.
- `eventio`: `kqueue` now ignores `KeyError` on `unregister`.
- `Redis`: `Message.reject` now supports the `requeue` argument.
- `Redis`: Remove superfluous pipeline call.

Fix contributed by Thomas Johansson.

- `Redis`: Now sets redelivered header for redelivered messages.
- Now always makes sure references to `sys.exc_info()` is removed.
- `Virtual`: The compression header is now removed before restoring messages.
- More tests for the SQLAlchemy backend.

Contributed by Franck Cuny.

- `Url` parsing did not handle MongoDB URLs properly.

Fix contributed by Flavio Percoco Premoli.

- `Beanstalk`: Ignore default tube when reserving.

Fix contributed by Zhao Xiaohong.

### 5.77.4 Nonblocking consume support

`librabbitmq`, `amqplib` and `redis` transports can now be used non-blocking.

The interface is very manual, and only consuming messages is non-blocking so far.

The API should not be regarded as stable or final in any way. It is used by Celery which has very limited needs at this point. Hopefully we can introduce a proper callback-based API later.

- `Transport.eventmap`

Is a map of `fd -> callback(fileno, event)` to register in an eventloop.

- `Transport.on_poll_start()`

Is called before every call to `poll`. The poller must support `register(fd, callback)` and `unregister(fd)` methods.

- `Transport.on_poll_start(poller)`

Called when the hub is initialized. The `poller` argument must support the same interface as `kombu.utils.eventio.poll`.

- `Connection.ensure_connection` now takes a callback argument which is called for every loop while the connection is down.

- Adds `connection.drain_nowait`

This is a non-blocking alternative to `drain_events`, but only supported by `amqplib/librabbitmq`.

- `drain_events` now sets `connection.more_to_read` if there is more data to read.

This is to support eventloops where other things must be handled between draining events.

## 5.78 2.1.8

**release-date** 2012-05-06 03:06 P.M BST

**release-by** Ask Solem

- Bound Exchange/Queue's are now pickleable.
- Consumer/Producer can now be instantiated without a channel, and only later bound using `.revive(channel)`.
- `ProducerPool` now takes `Producer` argument.
- `fxrange()` now counts forever if the stop argument is set to `None`. (`fxrange` is like `xrange` but for decimals).
- Auto delete support for virtual transports were incomplete and could lead to problems so it was removed.
- Cached declarations (`maybe_declare()`) are now bound to the underlying connection, so that entities are redeclared if the connection is lost.

This also means that previously uncacheable entities (e.g. non-durable) can now be cached.

- `compat ConsumerSet`: can now specify channel.

## 5.79 2.1.7

**release-date** 2012-04-27 06:00 P.M BST

**release-by** Ask Solem

- `compat consumerset` now accepts optional channel argument.

## 5.80 2.1.6

**release-date** 2012-04-23 01:30 P.M BST

**release-by** Ask Solem

- SQLAlchemy transport was not working correctly after URL parser change.
- `maybe_declare` now stores cached declarations per underlying connection instead of globally, in the rare case that data disappears from the broker after connection loss.
- Django: Added South migrations.

Contributed by Joseph Crosland.

## 5.81 2.1.5

**release-date** 2012-04-13 03:30 P.M BST

**release-by** Ask Solem

- The url parser removed more than the first leading slash (Issue #121).
- SQLAlchemy: Can now specify url using + separator

Example:

```
Connection('sqla+mysql://localhost/db')
```

- Better support for anonymous queues (Issue #116).  
Contributed by Michael Barrett.
- `Connection.as_uri` now quotes url parts (Issue #117).
- Beanstalk: Can now set message TTR as a message property.

Contributed by Andrii Kostenko

## 5.82 2.1.4

**release-date** 2012-04-03 04:00 P.M GMT

**release-by** Ask Solem

- MongoDB: URL parsing are now delegated to the pymongo library (Fixes Issue #103 and Issue #87).  
Fix contributed by Flavio Percoco Premoli and James Sullivan
- SQS: A bug caused SimpleDB to be used even if sdb persistence was not enabled (Issue #108).  
Fix contributed by Anand Kumria.
- Django: Transaction was committed in the wrong place, causing data cleanup to fail (Issue #115).  
Fix contributed by Daisuke Fujiwara.
- MongoDB: Now supports replica set URLs.  
Contributed by Flavio Percoco Premoli.
- Redis: Now raises a channel error if a queue key that is currently being consumed from disappears.  
Fix contributed by Stephan Jaekel.
- All transport 'channel\_errors' lists now includes `kombu.exception.StdChannelError`.
- All kombu exceptions now inherit from a common `KombuError`.

## 5.83 2.1.3

**release-date** 2012-03-20 03:00 P.M GMT

**release-by** Ask Solem

- Fixes Jython compatibility issues.
- Fixes Python 2.5 compatibility issues.

## 5.84 2.1.2

**release-date** 2012-03-01 01:00 P.M GMT

**release-by** Ask Solem

- amqplib: Last version broke SSL support.

## 5.85 2.1.1

**release-date** 2012-02-24 02:00 P.M GMT

**release-by** Ask Solem

- Connection URLs now supports encoded characters.
- Fixed a case where connection pool could not recover from connection loss.  
Fix contributed by Florian Munz.
- We now patch amqplib's `__del__` method to skip trying to close the socket if it is not connected, as this resulted in an annoying warning.
- Compression can now be used with binary message payloads.

Fix contributed by Steeve Morin.

## 5.86 2.1.0

**release-date** 2012-02-04 10:38 P.M GMT

**release-by** Ask Solem

- MongoDB: Now supports fanout (broadcast) (Issue #98).

Contributed by Scott Lyons.

- amqplib: Now detects broken connections by using `MSG_PEEK`.
- pylibrabbitmq: Now supports `basic_get` (Issue #97).
- gevent: Now always uses the `select` polling backend.
- pika transport: Now works with pika 0.9.5 and 0.9.6dev.

The old pika transport (supporting 0.5.x) is now available as alias `oldpika`.

(Note terribly latency has been experienced with the new pika versions, so this is still an experimental transport).

- Virtual transports: can now set polling interval via the transport options (Issue #96).

Example:

```
>>> Connection('sqs://', transport_options={
...     'polling_interval': 5.0})
```

The default interval is transport specific, but usually 1.0s (or 5.0s for the Django database transport, which can also be set using the `KOMBU_POLLING_INTERVAL` setting).

- Adds convenience function: `kombu.common.eventloop()`.



## 5.87 2.0.0

**release-date** 2012-01-15 06:34 P.M GMT

**release-by** Ask Solem

### 5.87.1 Important Notes

#### Python Compatibility

- No longer supports Python 2.4.

Users of Python 2.4 can still use the 1.x series.

The 1.x series has entered bugfix-only maintenance mode, and will stay that way as long as there is demand, and a willingness to maintain it.

#### New Transports

- `django-kombu` is now part of Kombu core.

The Django message transport uses the Django ORM to store messages.

It uses polling, with a default polling interval of 5 seconds. The polling interval can be increased or decreased by configuring the `KOMBU_POLLING_INTERVAL` Django setting, which is the polling interval in seconds as an int or a float. Note that shorter polling intervals can cause extreme strain on the database: if responsiveness is needed you shall consider switching to a non-polling transport.

To use it you must use transport alias "django", or as an URL:

```
django://
```

and then add `kombu.transport.django` to `INSTALLED_APPS`, and run `manage.py syncdb` to create the necessary database tables.

#### Upgrading

If you have previously used `django-kombu`, then the entry in `INSTALLED_APPS` must be changed from `django-kombu` to `kombu.transport.django`:

```
INSTALLED_APPS = (...,  
                  'kombu.transport.django')
```

If you have previously used `django-kombu`, then there is no need to recreate the tables, as the old tables will be fully compatible with the new version.

- `kombu-sqlalchemy` is now part of Kombu core.

This change requires no code changes given that the `sqlalchemy` transport alias is used.

### 5.87.2 News

- `kombu.mixins.ConsumerMixin` is a mixin class that lets you easily write consumer programs and threads.

See *Examples* and *Consumers*.

- SQS Transport: Added support for SQS queue prefixes (Issue #84).

The queue prefix can be set using the transport option `queue_name_prefix`:

```
BrokerTransport('SQS://', transport_options={
    'queue_name_prefix': 'myapp'})
```

Contributed by Nitzan Miron.

- `Producer.publish` now supports automatic retry.

Retry is enabled by the `reply` argument, and retry options set by the `retry_policy` argument:

```
exchange = Exchange('foo')
producer.publish(message, exchange=exchange, retry=True,
                 declare=[exchange], retry_policy={
                     'interval_start': 1.0})
```

See `ensure()` for a list of supported retry policy options.

- `Producer.publish` now supports a `declare` keyword argument.

This is a list of entities (`Exchange`, or `Queue`) that should be declared before the message is published.

### 5.87.3 Fixes

- Redis transport: Timeout was multiplied by 1000 seconds when using `select` for event I/O (Issue #86).

## 5.88 1.5.1

**release-date** 2011-11-30 01:00 P.M GMT

**release-by** Ask Solem

- Fixes issue with `kombu.compat` introduced in 1.5.0 (Issue #83).
- Adds the ability to disable `content_types` in the serializer registry.

Any message with a content type that is disabled will be refused. One example would be to disable the Pickle serializer:

```
>>> from kombu.serialization import registry
# by name
>>> registry.disable('pickle')
# or by mime-type.
>>> registry.disable('application/x-python-serialize')
```

## 5.89 1.5.0

**release-date** 2011-11-27 06:00 P.M GMT

**release-by** Ask Solem

- `kombu.pools`: Fixed a bug resulting in resources not being properly released.

This was caused by the use of `__hash__` to distinguish them.

- Virtual transports: Dead-letter queue is now disabled by default.

The dead-letter queue was enabled by default to help application authors, but now that Kombu is stable it should be removed. There are after all many cases where messages should just be dropped when there are no queues to buffer them, and keeping them without supporting automatic cleanup is rather considered a resource leak than a feature.

If wanted the dead-letter queue can still be enabled, by using the `deadletter_queue` transport option:

```
>>> x = Connection('redis://',
...                transport_options={'deadletter_queue': 'ae.undeliver'})
```

In addition, an `UndeliverableWarning` is now emitted when the dead-letter queue is enabled and a message ends up there.

Contributed by Ionel Maries Cristian.

- MongoDB transport now supports Replicasets (Issue #81).  
Contributed by Ivan Metzlar.
- The `Connection.ensure` methods now accepts a `max_retries` value of 0.  
A value of 0 now means *do not retry*, which is distinct from `None` which means *retry indefinitely*.  
Contributed by Dan McGee.
- SQS Transport: Now has a lowercase `sqs` alias, so that it can be used with broker URLs (Issue #82).  
Fix contributed by Hong Minhee
- SQS Transport: Fixes `KeyError` on message acknowledgements (Issue #73).  
The SQS transport now uses `UUID`'s for delivery tags, rather than a counter.  
Fix contributed by Brian Bernstein.
- SQS Transport: Unicode related fixes (Issue #82).  
Fix contributed by Hong Minhee.
- Redis version check could crash because of improper handling of types (Issue #63).
- Fixed error with `Resource.force_close_all` when resources were not yet properly initialized (Issue #78).

## 5.90 1.4.3

**release-date** 2011-10-27 10:00 P.M BST

**release-by** Ask Solem

- Fixes bug in `ProducerPool` where too many resources would be acquired.

## 5.91 1.4.2

**release-date** 2011-10-26 05:00 P.M BST

**release-by** Ask Solem

- Eventio: Polling should ignore `errno.EINTR`
- SQS: `str.encode` did only start accepting kwargs after Py2.7.

- `simple_task_queue` example didn't run correctly (Issue #72).  
Fix contributed by Stefan Eletzhofer.
- Empty messages would not raise an exception not able to be handled by `on_decode_error` (Issue #72)  
Fix contributed by Christophe Chauvet.
- CouchDB: Properly authenticate if user/password set (Issue #70)  
Fix contributed by Rafael Duran Castaneda
- `Connection.Consumer` had the wrong signature.  
Fix contributed by Pavel Skvazh

## 5.92 1.4.1

**release-date** 2011-09-26 04:00 P.M BST

**release-by** Ask Solem

- 1.4.0 broke the producer pool, resulting in new connections being established for every acquire.

## 5.93 1.4.0

**release-date** 2011-09-22 05:00 P.M BST

**release-by** Ask Solem

- Adds module `kombu.mixins`.  
This module contains a `ConsumerMixin` class that can be used to easily implement a message consumer thread that consumes messages from one or more `kombu.Consumer` instances.
- New example: *Task Queue Example*  
Using the `ConsumerMixin`, default channels and the global connection pool to demonstrate new Kombu features.
- MongoDB transport did not work with MongoDB  $\geq 2.0$  (Issue #66)  
Fix contributed by James Turk.
- Redis-py version check did not account for beta identifiers in version string.  
Fix contributed by David Ziegler.
- Producer and Consumer now accepts a connection instance as the first argument.  
The connections default channel will then be used.  
In addition shortcut methods has been added to `Connection`:

```
>>> connection.Producer(exchange)
>>> connection.Consumer(queues=..., callbacks=...)
```

- `Connection` has acquired a `connected` attribute that can be used to check if the connection instance has established a connection.
- `ConnectionPool.acquire_channel` now returns the connections default channel rather than establishing a new channel that must be manually handled.

- Added `kombu.common.maybe_declare`  
`maybe_declare(entity)` declares an entity if it has not previously been declared in the same process.
- `kombu.compat.entry_to_queue()` has been moved to `kombu.common`
- New module `kombu.clocks` now contains an implementation of Lamports logical clock.

## 5.94 1.3.5

**release-date** 2011-09-16 06:00 P.M BST

**release-by** Ask Solem

- Python 3: `AMQP_PROTOCOL_HEADER` must be bytes, not str.

## 5.95 1.3.4

**release-date** 2011-09-16 06:00 P.M BST

**release-by** Ask Solem

- Fixes syntax error in `pools.reset`

## 5.96 1.3.3

**release-date** 2011-09-15 02:00 P.M BST

**release-by** Ask Solem

- `pools.reset` did not support after forker arguments.

## 5.97 1.3.2

**release-date** 2011-09-10 01:00 P.M BST

**release-by** Mher Movsisyan

- Broke Python 2.5 compatibility by importing `parse_qs` from `urlparse`
- `Connection.default_channel` is now closed when connection is revived after connection failures.
- Pika: Channel now supports the `connection.client` attribute as required by the simple interface.
- `pools.set_limit` now raises an exception if the limit is lower than the previous limit.
- `pools.set_limit` no longer resets the pools.

## 5.98 1.3.1

**release-date** 2011-10-07 03:00 P.M BST

**release-by** Ask Solem

- Last release broke after fork for pool reinitialization.
- Producer/Consumer now has a `connection` attribute, giving access to the `Connection` of the instance.
- Pika: Channels now have access to the underlying `Connection` instance using `channel.connection.client`.

This was previously required by the `Simple` classes and is now also required by `Consumer` and `Producer`.

- `Connection.default_channel` is now closed at object revival.
- Adds `kombu.clocks.LamportClock`.
- `compat.entry_to_queue` has been moved to new module `kombu.common`.

## 5.99 1.3.0

**release-date** 2011-10-05 01:00 P.M BST

**release-by** Ask Solem

- Broker connection info can now be specified using URLs

The broker hostname can now be given as an URL instead, of the format:

```
transport://user:password@hostname:port/virtual_host
```

for example the default broker is expressed as:

```
>>> Connection('amqp://guest:guest@localhost:5672//')
```

Transport defaults to `amqp`, and is not required. `user`, `password`, `port` and `virtual_host` is also not mandatory and will default to the corresponding transports default.

---

**Note:** Note that the path component (`virtual_host`) always starts with a forward-slash. This is necessary to distinguish between the virtual host "" (empty) and '/', which are both acceptable virtual host names.

A virtual host of '/' becomes:

```
amqp://guest:guest@localhost:5672//
```

and a virtual host of "" (empty) becomes:

```
amqp://guest:guest@localhost:5672/
```

---

So the leading slash in the path component is **always required**.

---

- Now comes with default global connection and producer pools.

The acquire a connection using the connection parameters from a `Connection`:

```
>>> from kombu import Connection, connections
>>> connection = Connection('amqp://guest:guest@localhost//')
>>> with connections[connection].acquire(block=True):
...     # do something with connection
```

To acquire a producer using the connection parameters from a Connection:

```
>>> from kombu import Connection, producers
>>> connection = Connection('amqp://guest:guest@localhost//')
>>> with producers[connection].acquire(block=True):
...     producer.publish({'hello': 'world'}, exchange='hello')
```

Acquiring a producer will in turn also acquire a connection from the associated pool in connections, so you the number of producers is bound the same limit as number of connections.

The default limit of 100 connections per connection instance can be changed by doing:

```
>>> from kombu import pools
>>> pools.set_limit(10)
```

The pool can also be forcefully closed by doing:

```
>>> from kombu import pools
>>> pool.reset()
```

- SQS Transport: Persistence using SimpleDB is now disabled by default, after reports of unstable SimpleDB connections leading to errors.
- Producer can now be used as a context manager.
- Producer.\_\_exit\_\_ now properly calls release instead of close.

The previous behavior would lead to a memory leak when using the `kombu.pools.ProducerPool`

- Now silences all exceptions from `import ctypes` to match behaviour of the standard Python `uuid` module, and avoid passing on MemoryError exceptions on SELinux-enabled systems (Issue #52 + Issue #53)
- `amqp` is now an alias to the `amqplib` transport.
- `kombu.syn.detect_environment` now returns 'default', 'eventlet', or 'gevent' depending on what monkey patches have been installed.
- Serialization registry has new attribute `type_to_name` so it is possible to lookup serializater name by content type.
- Exchange argument to `Producer.publish` can now be an Exchange instance.
- `compat.Publisher` now supports the `channel` keyword argument.
- Acking a message on some transports could lead to `KeyError` being raised (Issue #57).
- Connection pool: Connections are no long instantiated when the pool is created, but instantiated as needed instead.
- Tests now pass on PyPy.
- `Connection.as_uri` now includes the password if the keyword argument `include_password` is set.
- Virtual transports now comes with a default `default_connection_params` attribute.

## 5.100 1.2.1

**release-date** 2011-07-29 12:52 P.M BST

**release-by** Ask Solem

- Now depends on amqplib  $\geq$  1.0.0.
- Redis: Now automatically deletes `auto_delete` queues at `basic_cancel`.
- `serialization.unregister` added so it is possible to remove unwanted serializers.
- Fixes `MemoryError` while importing `ctypes` on SELinux (Issue #52).
- `Connection.autoretry` is a version of `ensure` that works with arbitrary functions (i.e. it does not need an associated object that implements the `revive` method).

Example usage:

```
channel = connection.channel()
try:
    ret, channel = connection.autoretry(send_messages, channel=channel)
finally:
    channel.close()
```

- `ConnectionPool.acquire` no longer force establishes the connection.

The connection will be established as needed.

- `Connection.ensure` now supports an `on_revive` callback that is applied whenever the connection is re-established.
- `Consumer.consuming_from(queue)` returns `True` if the `Consumer` is consuming from `queue`.
- `Consumer.cancel_by_queue` did not remove the queue from `queues`.
- `compat.ConsumerSet.add_queue_from_dict` now automatically declared the queue if `auto_declare` set.

## 5.101 1.2.0

**release-date** 2011-07-15 12:00 P.M BST

**release-by** Ask Solem

- Virtual: Fixes cyclic reference in `Channel.close` (Issue #49).
- `Producer.publish`: Can now set additional properties using keyword arguments (Issue #48).
- Adds `Queue.no_ack` option to control the `no_ack` option for individual queues.
- Recent versions broke `pylibrabbitmq` support.
- `SimpleQueue` and `SimpleBuffer` can now be used as contexts.
- Test requirements specifies `PyYAML==3.09` as 3.10 dropped Python 2.4 support
- Now properly reports default values in `Connection.info/as_uri`



## 5.102 1.1.6

**release-date** 2011-06-13 04:00 P.M BST

**release-by** Ask Solem

- Redis: Fixes issue introduced in 1.1.4, where a redis connection failure could leave consumer hanging forever.
- SQS: Now supports fanout messaging by using SimpleDB to store routing tables.

This can be disabled by setting the *supports\_fanout* transport option:

```
>>> Connection(transport='SQS',
...             transport_options={'supports_fanout': False})
```

- SQS: Now properly deletes a message when a message is acked.
- SQS: Can now set the Amazon AWS region, by using the *region* transport option.
- amqplib: Now uses *localhost* as default hostname instead of raising an error.

## 5.103 1.1.5

**release-date** 2011-06-07 06:00 P.M BST

**release-by** Ask Solem

- Fixes compatibility with redis-py 2.4.4.

## 5.104 1.1.4

**release-date** 2011-06-07 04:00 P.M BST

**release-by** Ask Solem

- Redis transport: Now requires redis-py version 2.4.4 or later.
- New Amazon SQS transport added.

Usage:

```
>>> conn = Connection(transport='SQS',
...                   userid=aws_access_key_id,
...                   password=aws_secret_access_key)
```

The environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are also supported.

- librabbitmq transport: Fixes default credentials support.
- amqplib transport: Now supports *login\_method* for SSL auth.

`Connection` now supports the *login\_method* keyword argument.

Default *login\_method* is `AMQPLAIN`.

## 5.105 1.1.3

**release-date** 2011-04-21 04:00 P.M CEST

**release-by** Ask Solem

- Redis: Consuming from multiple connections now works with Eventlet.
- Redis: Can now perform channel operations while the channel is in BRPOP/LISTEN mode (Issue #35).  
Also the async BRPOP now times out after 1 second, this means that cancelling consuming from a queue/starting consuming from additional queues has a latency of up to one second (BRPOP does not support subsecond timeouts).
- Virtual: Allow channel objects to be closed multiple times without error.
- amqplib: `AttributeError` has been added to the list of known connection related errors (`Connection.connection_errors`).
- amqplib: Now converts `SSLerror` timeout errors to `socket.timeout` (<http://bugs.python.org/issue10272>)
- Ensures cyclic references are destroyed when the connection is closed.

## 5.106 1.1.2

**release-date** 2011-04-06 04:00 P.M CEST

**release-by** Ask Solem

- Redis: Fixes serious issue where messages could be lost.  
The issue could happen if the message exceeded a certain number of kilobytes in size.  
It is recommended that all users of the Redis transport should upgrade to this version, even if not currently experiencing any issues.

## 5.107 1.1.1

**release-date** 2011-04-05 03:51 P.M CEST

**release-by** Ask Solem

- 1.1.0 started using `Queue.LifoQueue` which is only available in Python 2.6+ (Issue #33). We now ship with our own `LifoQueue`.

## 5.108 1.1.0

**release-date** 2011-04-05 01:05 P.M CEST

**release-by** Ask Solem

## 5.108.1 Important Notes

- Virtual transports: Message body is now base64 encoded by default (Issue #27).

This should solve problems sending binary data with virtual transports.

Message compatibility is handled by adding a `body_encoding` property, so messages sent by older versions is compatible with this release. However – If you are accessing the messages directly not using Kombu, then you have to respect the `body_encoding` property.

If you need to disable base64 encoding then you can do so via the transport options:

```
Connection(transport='...',
            transport_options={'body_encoding': None})
```

### For transport authors:

You don't have to change anything in your custom transports, as this is handled automatically by the base class.

If you want to use a different encoder you can do so by adding a key to `Channel.codecs`. Default encoding is specified by the `Channel.body_encoding` attribute.

A new codec must provide two methods: `encode(data)` and `decode(data)`.

- `ConnectionPool/ChannelPool/Resource`: Setting `limit=None` (or 0) now disables pool semantics, and will establish and close the resource whenever acquired or released.
- `ConnectionPool/ChannelPool/Resource`: Is now using a LIFO queue instead of the previous FIFO behavior.
  - This means that the last resource released will be the one acquired next. I.e. if only a single thread is using the pool this means only a single connection will ever be used.
- `Connection`: Cloned connections did not inherit `transport_options` (`__copy__`).
- `contrib/requirements` is now located in the top directory of the distribution.
- `MongoDB`: Now supports authentication using the `userid` and `password` arguments to `Connection` (Issue #30).
- `Connection`: Default authentication credentials are now delegated to the individual transports.

This means that the `userid` and `password` arguments to `Connection` is no longer *guest/guest* by default.

The `amqplib` and `pika` transports will still have the default credentials.

- `Consumer.__exit__()` did not have the correct signature (Issue #32).
- `Channel` objects now have a `channel_id` attribute.
- **MongoDB: Version sniffing broke with development versions of mongod** (Issue #29).
- **New environment variable `KOMBU_LOG_CONNECTION` will now emit debug** log messages for connection related actions.

`KOMBU_LOG_DEBUG` will also enable `KOMBU_LOG_CONNECTION`.

## 5.109 1.0.7

release-date 2011-03-28 05:45 P.M CEST

**release-by** Ask Solem

- Now depends on anyjson 0.3.1
  - cjson is no longer a recommended json implementation, and anyjson will now emit a deprecation warning if used.
- Please note that the Pika backend only works with version 0.5.2.
  - The latest version (0.9.x) drastically changed API, and it is not compatible yet.
- `on_decode_error` is now called for exceptions in `message_to_python` (Issue #24).
- Redis: did not respect QoS settings.
- Redis: Creating a connection now ensures the connection is established.
  - This means `Connection.ensure_connection` works properly with Redis.
- `consumer_tag` argument to `Queue.consume` can't be `None` (Issue #21).
  - A `None` value is now automatically converted to empty string. An empty string will make the server generate a unique tag.
- Connection now supports a `transport_options` argument.
  - This can be used to pass additional arguments to transports.
- Pika: `drain_events` raised `socket.timeout` even if no timeout set (Issue #8).

## 5.110 1.0.6

**release-date** 2011-03-22 04:00 P.M CET

**release-by** Ask Solem

- The `delivery_mode` aliases (persistent/transient) were not automatically converted to integer, and would cause a crash if using the amqplib transport.
- Redis: The redis-py `InvalidData` exception suddenly changed name to `DataError`.
- The `KOMBU_LOG_DEBUG` environment variable can now be set to log all channel method calls.

Support for the following environment variables have been added:

- `KOMBU_LOG_CHANNEL` will wrap channels in an object that logs every method call.
- `KOMBU_LOG_DEBUG` both enables channel logging and configures the root logger to emit messages to standard error.

**Example Usage:**

```
$ KOMBU_LOG_DEBUG=1 python
>>> from kombu import Connection
>>> conn = Connection()
>>> channel = conn.channel()
Start from server, version: 8.0, properties:
  {u'product': 'RabbitMQ',..... }
Open OK! known_hosts []
using channel_id: 1
Channel open
>>> channel.queue_declare('myq', passive=True)
[Kombu channel:1] queue_declare('myq', passive=True)
(u'myq', 0, 1)
```

## 5.111 1.0.5

**release-date** 2011-03-17 04:00 P.M CET

**release-by** Ask Solem

- Fixed memory leak when creating virtual channels. All virtual transports affected (redis, mongodb, memory, django, sqlalchemy, couchdb, beanstalk).
- Virtual Transports: Fixed potential race condition when acking messages.
  - If you have been affected by this, the error would show itself as an exception raised by the Ordered-Dict implementation. (`object no longer exists`).
- MongoDB transport requires the `findandmodify` command only available in MongoDB 1.3+, so now raises an exception if connected to an incompatible server version.
- Virtual Transports: `basic.cancel` should not try to remove unknown consumer tag.

## 5.112 1.0.4

**release-date** 2011-02-28 04:00 P.M CET

**release-by** Ask Solem

- Added `Transport.polling_interval`
  - Used by `django-kombu` to increase the time to sleep between `SELECT`s when there are no messages in the queue.
  - Users of `django-kombu` should upgrade to `django-kombu v0.9.2`.

## 5.113 1.0.3

**release-date** 2011-02-12 04:00 P.M CET

**release-by** Ask Solem

- `ConnectionPool`: Re-connect if `amqplib` connection closed
- Adds `Queue.as_dict + Exchange.as_dict`.
- Copyright headers updated to include 2011.

## 5.114 1.0.2

**release-date** 2011-01-31 10:45 P.M CET

**release-by** Ask Solem

- `amqplib`: Message properties were not set properly.
- Ghettoq backend names are now automatically translated to the new names.

## 5.115 1.0.1

**release-date** 2011-01-28 12:00 P.M CET

**release-by** Ask Solem

- Redis: Now works with Linux (epoll)

## 5.116 1.0.0

**release-date** 2011-01-27 12:00 P.M CET

**release-by** Ask Solem

- Initial release

## 5.117 0.1.0

**release-date** 2010-07-22 04:20 P.M CET

**release-by** Ask Solem

- Initial fork of carrot

---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`





**k**

- kombu, 31
- kombu.abstract, 75
- kombu.async, 76
- kombu.async.debug, 80
- kombu.async.hub, 77
- kombu.async.semaphore, 78
- kombu.async.timer, 79
- kombu.clocks, 53
- kombu.common, 48
- kombu.compat, 55
- kombu.compression, 72
- kombu.connection, 64
- kombu.exceptions, 63
- kombu.five, 164
- kombu.log, 63
- kombu.message, 71
- kombu.mixins, 50
- kombu.pidbox, 61
- kombu.pools, 73
- kombu.serialization, 157
- kombu.simple, 51
- kombu.syn, 76
- kombu.transport, 81
- kombu.transport.amqpplib, 148
- kombu.transport.base, 150
- kombu.transport.beanstalk, 138
- kombu.transport.couchdb, 140
- kombu.transport.django, 143
- kombu.transport.django.management.commands.clean\_kombu\_messages, 145
- kombu.transport.django.managers, 145
- kombu.transport.django.models, 144
- kombu.transport.filesystem, 142
- kombu.transport.memory, 133
- kombu.transport.mongodb, 139
- kombu.transport.pyamqp, 81
- kombu.transport.pyro, 147
- kombu.transport.qpid, 99
- kombu.transport.redis, 134
- kombu.transport.SLMQ, 146
- kombu.transport.virtual, 151
- kombu.transport.virtual.exchange, 156
- kombu.transport.virtual.scheduling, 157
- kombu.transport.zmq, 138
- kombu.transport.zookeeper, 141
- kombu.utils, 159
- kombu.utils.amq\_manager, 164
- kombu.utils.compat, 162
- kombu.utils.debug, 162
- kombu.utils.encoding, 162
- kombu.utils.eventio, 161
- kombu.utils.functional, 163
- kombu.utils.limits, 161
- kombu.utils.text, 164
- kombu.utils.url, 163



## Symbols

`__len__()` (kombu.simple.SimpleBuffer method), 53  
`__len__()` (kombu.simple.SimpleQueue method), 52  
`_close()` (kombu.Connection method), 36

## A

`abcast()` (kombu.pidbox.Mailbox method), 62  
 AbstractChannel (class in kombu.transport.virtual), 152  
`accept` (kombu.compat.Consumer attribute), 56  
`accept` (kombu.compat.ConsumerSet attribute), 59  
`accept` (kombu.message.Message attribute), 71  
`accept` (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83  
`accept` (kombu.transport.virtual.Message attribute), 154  
`ack()` (kombu.message.Message method), 71  
`ack()` (kombu.transport.base.Message method), 150  
`ack()` (kombu.transport.pyamqp.Connection.Channel.Message method), 83  
`ack()` (kombu.transport.qpid.Channel.QoS method), 125  
`ack()` (kombu.transport.qpid.Connection.Channel.QoS method), 116  
`ack()` (kombu.transport.qpid.Transport.Connection.Channel.QoS method), 103  
`ack()` (kombu.transport.redis.Channel.QoS method), 136  
`ack()` (kombu.transport.redis.Transport.Channel.QoS method), 135  
`ack()` (kombu.transport.virtual.Message method), 154  
`ack()` (kombu.transport.virtual.QoS method), 155  
`ack_emulation` (kombu.transport.redis.Channel attribute), 137  
`ack_emulation` (kombu.transport.redis.Transport.Channel attribute), 135  
`ack_log_error()` (kombu.message.Message method), 71  
`ack_log_error()` (kombu.transport.pyamqp.Connection.Channel.Message method), 83  
`ack_log_error()` (kombu.transport.virtual.Message method), 154  
`acknowledged` (kombu.message.Message attribute), 71  
`acknowledged` (kombu.transport.base.Message attribute), 150  
`acknowledged` (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83  
`acknowledged` (kombu.transport.virtual.Message attribute), 154  
`acquire()` (kombu.async.semaphore.LaxBoundedSemaphore method), 79  
`acquire()` (kombu.connection.ChannelPool method), 71  
`acquire()` (kombu.connection.ConnectionPool method), 70  
`active_queues` (kombu.transport.redis.Channel attribute), 137  
`active_queues` (kombu.transport.redis.Transport.Channel attribute), 135  
`add()` (kombu.async.Hub method), 76  
`add()` (kombu.async.hub.Hub method), 77  
`add_consumer()` (kombu.compat.ConsumerSet method), 59  
`add_consumer_from_dict()` (kombu.compat.ConsumerSet method), 59  
`add_queue()` (kombu.compat.Consumer method), 56  
`add_queue()` (kombu.compat.ConsumerSet method), 59  
`add_queue()` (kombu.Consumer method), 47  
`add_queue_from_dict()` (kombu.compat.Consumer method), 57  
`add_queue_from_dict()` (kombu.compat.ConsumerSet method), 59  
`add_queue_from_dict()` (kombu.Consumer method), 47  
`add_reader()` (kombu.async.Hub method), 76  
`add_reader()` (kombu.async.hub.Hub method), 77  
`add_writer()` (kombu.async.Hub method), 76  
`add_writer()` (kombu.async.hub.Hub method), 77  
`adjust()` (kombu.clocks.LamportClock method), 54  
`after_reply_message_received()` (kombu.transport.memory.Channel method), 134  
`after_reply_message_received()` (kombu.transport.memory.Transport.Channel method), 134  
`after_reply_message_received()` (kombu.transport.pyamqp.Connection.Channel method), 84

- after\_reply\_message\_received() (kombu.transport.pyro.Channel method), 148
  - after\_reply\_message\_received() (kombu.transport.pyro.Transport.Channel method), 147
  - alias (kombu.Queue attribute), 42
  - AMQP\_PROTOCOL\_HEADER (kombu.transport.amqplib.Connection attribute), 149
  - annotate() (kombu.log.LogMixin method), 63
  - append() (kombu.transport.qpid.Channel.QoS method), 126
  - append() (kombu.transport.qpid.Connection.Channel.QoS method), 116
  - append() (kombu.transport.qpid.Transport.Connection.Channel.QoS method), 103
  - append() (kombu.transport.redis.Channel.QoS method), 136
  - append() (kombu.transport.redis.Transport.Channel.QoS method), 135
  - append() (kombu.transport.virtual.QoS method), 155
  - apply\_entry() (kombu.async.timer.Timer method), 80
  - args (kombu.async.timer.Entry attribute), 79
  - args (kombu.async.timer.Timer.Entry attribute), 80
  - args (kombu.compat.Consumer.ContentDisallowed attribute), 56
  - args (kombu.compat.ConsumerSet.ContentDisallowed attribute), 59
  - args (kombu.transport.pyamqp.Connection.Channel.Message.MessageStateError attribute), 83
  - args (kombu.transport.virtual.Message.MessageStateError attribute), 154
  - arguments (kombu.Exchange attribute), 39
  - as\_dict() (kombu.Queue method), 42
  - as\_uri() (kombu.Connection method), 34
  - as\_uri() (kombu.connection.Connection method), 67
  - as\_url() (in module kombu.utils.url), 163
  - async\_pool (kombu.transport.redis.Channel attribute), 137
  - async\_pool (kombu.transport.redis.Transport.Channel attribute), 135
  - attrs (kombu.common.Broadcast attribute), 48
  - attrs (kombu.Exchange attribute), 40
  - attrs (kombu.Queue attribute), 42
  - auto\_declare (kombu.compat.Consumer attribute), 57
  - auto\_declare (kombu.compat.ConsumerSet attribute), 59
  - auto\_declare (kombu.compat.Publisher attribute), 55
  - auto\_declare (kombu.Consumer attribute), 46
  - auto\_declare (kombu.pools.ProducerPool.Producer attribute), 74
  - auto\_declare (kombu.Producer attribute), 44
  - auto\_delete (kombu.compat.Consumer attribute), 57
  - auto\_delete (kombu.compat.Publisher attribute), 55
  - auto\_delete (kombu.Exchange attribute), 39, 40
  - auto\_delete (kombu.Queue attribute), 42
  - autoretry() (kombu.Connection method), 34
  - autoretry() (kombu.connection.Connection method), 67
  - AWS\_ACCESS\_KEY\_ID, 209
  - AWS\_SECRET\_ACCESS\_KEY, 209
- ## B
- backend (kombu.compat.Publisher attribute), 55
  - basic\_ack() (kombu.transport.pyamqp.Connection.Channel method), 84
  - basic\_ack() (kombu.transport.qpid.Channel method), 127
  - basic\_ack() (kombu.transport.qpid.Connection.Channel method), 117
  - basic\_ack() (kombu.transport.qpid.Transport.Connection.Channel method), 104
  - basic\_ack() (kombu.transport.SLMQ.Channel method), 146
  - basic\_ack() (kombu.transport.SLMQ.Transport.Channel method), 146
  - basic\_ack() (kombu.transport.virtual.Channel method), 153
  - basic\_cancel() (kombu.transport.amqplib.Channel method), 149
  - basic\_cancel() (kombu.transport.pyamqp.Connection.Channel method), 84
  - basic\_cancel() (kombu.transport.qpid.Channel method), 127
  - basic\_cancel() (kombu.transport.qpid.Connection.Channel method), 117
  - basic\_cancel() (kombu.transport.qpid.Transport.Connection.Channel method), 104
  - basic\_cancel() (kombu.transport.redis.Channel method), 137
  - basic\_cancel() (kombu.transport.redis.Transport.Channel method), 135
  - basic\_cancel() (kombu.transport.SLMQ.Channel method), 147
  - basic\_cancel() (kombu.transport.SLMQ.Transport.Channel method), 146
  - basic\_cancel() (kombu.transport.virtual.Channel method), 153
  - basic\_consume() (kombu.transport.amqplib.Channel method), 149
  - basic\_consume() (kombu.transport.django.Channel method), 144
  - basic\_consume() (kombu.transport.django.Transport.Channel method), 143
  - basic\_consume() (kombu.transport.pyamqp.Connection.Channel method), 85
  - basic\_consume() (kombu.transport.qpid.Channel method), 127
  - basic\_consume() (kombu.transport.qpid.Connection.Channel method), 118

- basic\_consume() (kombu.transport.qpid.Transport.Connection.Channel method), 105
  - basic\_consume() (kombu.transport.redis.Channel method), 137
  - basic\_consume() (kombu.transport.redis.Transport.Channel method), 135
  - basic\_consume() (kombu.transport.SLMQ.Channel method), 147
  - basic\_consume() (kombu.transport.SLMQ.Transport.Channel method), 146
  - basic\_consume() (kombu.transport.virtual.Channel method), 153
  - basic\_get() (kombu.transport.pyamqp.Connection.Channel method), 86
  - basic\_get() (kombu.transport.qpid.Channel method), 128
  - basic\_get() (kombu.transport.qpid.Connection.Channel method), 118
  - basic\_get() (kombu.transport.qpid.Transport.Connection.Channel method), 105
  - basic\_get() (kombu.transport.virtual.Channel method), 153
  - basic\_publish() (kombu.transport.pyamqp.Connection.Channel method), 86
  - basic\_publish() (kombu.transport.qpid.Channel method), 128
  - basic\_publish() (kombu.transport.qpid.Connection.Channel method), 119
  - basic\_publish() (kombu.transport.qpid.Transport.Connection.Channel method), 106
  - basic\_publish() (kombu.transport.virtual.Channel method), 153
  - basic\_publish\_confirm() (kombu.transport.pyamqp.Connection.Channel method), 87
  - basic\_qos() (kombu.transport.pyamqp.Connection.Channel method), 87
  - basic\_qos() (kombu.transport.qpid.Channel method), 128
  - basic\_qos() (kombu.transport.qpid.Connection.Channel method), 119
  - basic\_qos() (kombu.transport.qpid.Transport.Connection.Channel method), 106
  - basic\_qos() (kombu.transport.virtual.Channel method), 153
  - basic\_recover() (kombu.transport.pyamqp.Connection.Channel method), 88
  - basic\_recover() (kombu.transport.virtual.Channel method), 153
  - basic\_recover\_async() (kombu.transport.pyamqp.Connection.Channel method), 88
  - basic\_reject() (kombu.transport.pyamqp.Connection.Channel method), 88
  - basic\_reject() (kombu.transport.qpid.Channel method), 129
  - basic\_reject() (kombu.transport.qpid.Connection.Channel method), 119
  - basic\_reject() (kombu.transport.qpid.Transport.Connection.Channel method), 106
  - basic\_reject() (kombu.transport.virtual.Channel method), 153
  - bind() (kombu.abstract.MaybeChannelBound method), 75
  - bind() (kombu.Queue method), 42
  - bind\_to() (kombu.Exchange method), 40
  - bind\_to() (kombu.Queue method), 42
  - binding() (kombu.Exchange method), 40
  - binding\_arguments (kombu.Queue attribute), 42
  - bindings (kombu.transport.virtual.BrokerState attribute), 156
  - body (kombu.message.Message attribute), 72
  - body (kombu.transport.base.Message attribute), 150
  - body (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83
  - body (kombu.transport.virtual.Message attribute), 154
  - body\_encoding (kombu.transport.qpid.Channel attribute), 129
  - body\_encoding (kombu.transport.qpid.Connection.Channel attribute), 120
  - body\_encoding (kombu.transport.qpid.Transport.Connection.Channel attribute), 107
  - Broadcast (class in kombu.common), 48
  - BrokerState (class in kombu.transport.virtual), 156
  - bytes\_to\_str() (in module kombu.utils.encoding), 163
- ## C
- cached\_property (class in kombu.utils), 160
  - call() (kombu.pidbox.Mailbox method), 62
  - call\_after() (kombu.async.timer.Timer method), 80
  - call\_at() (kombu.async.Hub method), 76
  - call\_at() (kombu.async.hub.Hub method), 77
  - call\_at() (kombu.async.timer.Timer method), 80
  - call\_later() (kombu.async.Hub method), 76
  - call\_later() (kombu.async.hub.Hub method), 77
  - call\_repeatedly() (kombu.async.Hub method), 76
  - call\_repeatedly() (kombu.async.hub.Hub method), 77
  - call\_repeatedly() (kombu.async.timer.Timer method), 80
  - call\_soon() (kombu.async.Hub method), 76
  - call\_soon() (kombu.async.hub.Hub method), 77
  - callback\_for() (in module kombu.async.debug), 80
  - callbacks (kombu.compat.Consumer attribute), 57
  - callbacks (kombu.compat.ConsumerSet attribute), 59
  - callbacks (kombu.Consumer attribute), 46
  - can\_cache\_declaration (kombu.abstract.MaybeChannelBound attribute), 75
  - can\_cache\_declaration (kombu.Exchange attribute), 40
  - can\_cache\_declaration (kombu.Queue attribute), 42
  - can\_consume() (kombu.transport.qpid.Channel.QoS method), 126
  - can\_consume() (kombu.transport.qpid.Connection.Channel.QoS method), 116

- can\_consume() (kombu.transport.qpid.Transport.Connection.Channel.QoS method), 104
- can\_consume() (kombu.transport.virtual.QoS method), 155
- can\_consume() (kombu.utils.limits.TokenBucket method), 161
- can\_consume\_max\_estimate() (kombu.transport.qpid.Channel.QoS method), 126
- can\_consume\_max\_estimate() (kombu.transport.qpid.Connection.Channel.QoS method), 117
- can\_consume\_max\_estimate() (kombu.transport.qpid.Transport.Connection.Channel.QoS method), 104
- can\_consume\_max\_estimate() (kombu.transport.virtual.QoS method), 155
- can\_parse\_url (kombu.transport.mongodb.Transport attribute), 139
- cancel() (kombu.async.timer.Entry method), 79
- cancel() (kombu.async.timer.Timer method), 80
- cancel() (kombu.async.timer.Timer.Entry method), 80
- cancel() (kombu.compat.Consumer method), 57
- cancel() (kombu.compat.ConsumerSet method), 59
- cancel() (kombu.Consumer method), 47
- cancel() (kombu.Queue method), 42
- cancel\_by\_queue() (kombu.compat.Consumer method), 57
- cancel\_by\_queue() (kombu.compat.ConsumerSet method), 59
- cancel\_by\_queue() (kombu.Consumer method), 47
- cancelled (kombu.async.timer.Entry attribute), 79
- cancelled (kombu.async.timer.Timer.Entry attribute), 80
- capacity (kombu.utils.limits.TokenBucket attribute), 161
- cast() (kombu.pidbox.Mailbox method), 62
- Channel (class in kombu.transport.amqpplib), 149
- Channel (class in kombu.transport.beanstalk), 139
- Channel (class in kombu.transport.couchdb), 141
- Channel (class in kombu.transport.django), 144
- Channel (class in kombu.transport.filesystem), 143
- Channel (class in kombu.transport.memory), 134
- Channel (class in kombu.transport.mongodb), 140
- Channel (class in kombu.transport.pyamqp), 99
- Channel (class in kombu.transport.pyro), 148
- Channel (class in kombu.transport.qpid), 124
- Channel (class in kombu.transport.redis), 136
- Channel (class in kombu.transport.SLMQ), 146
- Channel (class in kombu.transport.virtual), 152
- Channel (class in kombu.transport.zookeeper), 142
- channel (kombu.abstract.MaybeChannelBound attribute), 75
- channel (kombu.compat.Consumer attribute), 57
- channel (kombu.compat.ConsumerSet attribute), 59
- channel (kombu.compat.Publisher attribute), 55
- channel (kombu.Consumer attribute), 46
- channel (kombu.Exchange attribute), 39
- channel (kombu.message.Message attribute), 72
- channel (kombu.pidbox.Node attribute), 63
- channel (kombu.pools.ProducerPool.Producer attribute), 74
- channel (kombu.Producer attribute), 44
- channel (kombu.Queue attribute), 42
- channel (kombu.simple.SimpleBuffer attribute), 52
- channel (kombu.simple.SimpleQueue attribute), 52
- channel (kombu.transport.base.Message attribute), 150
- channel (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83
- channel (kombu.transport.virtual.Message attribute), 154
- Channel (kombu.transport.virtual.Transport attribute), 152
- channel() (kombu.Connection method), 34
- channel() (kombu.connection.Connection method), 67
- channel() (kombu.transport.amqpplib.Connection method), 149
- channel() (kombu.transport.amqpplib.Transport.Connection method), 148
- channel() (kombu.transport.pyamqp.Connection method), 98
- Channel.Message (class in kombu.transport.amqpplib), 149
- Channel.Message (class in kombu.transport.pyamqp), 99
- Channel.Message (class in kombu.transport.qpid), 125
- Channel.QoS (class in kombu.transport.qpid), 125
- Channel.QoS (class in kombu.transport.redis), 136
- channel\_errors (kombu.Connection attribute), 33
- channel\_errors (kombu.connection.Connection attribute), 67
- channel\_errors (kombu.mixins.ConsumerMixin attribute), 51
- channel\_errors (kombu.transport.amqpplib.Transport attribute), 148
- channel\_errors (kombu.transport.base.Transport attribute), 151
- channel\_errors (kombu.transport.beanstalk.Transport attribute), 138
- channel\_errors (kombu.transport.couchdb.Transport attribute), 141
- channel\_errors (kombu.transport.django.Transport attribute), 143
- channel\_errors (kombu.transport.mongodb.Transport attribute), 139
- channel\_errors (kombu.transport.pyamqp.Connection attribute), 98
- channel\_errors (kombu.transport.pyamqp.Transport attribute), 82
- channel\_errors (kombu.transport.qpid.Transport attribute), 112
- channel\_errors (kombu.transport.zookeeper.Transport attribute), 142

- tribute), 142
- ChannelLimitExceeded, 63
- ChannelPool (class in kombu.connection), 71
- ChannelPool() (kombu.Connection method), 37
- ChannelPool() (kombu.connection.Connection method), 65
- cleanup() (kombu.transport.django.managers.MessageManager method), 145
- cleanup\_every (kombu.transport.django.managers.MessageManager attribute), 145
- clear() (kombu.async.semaphore.LaxBoundedSemaphore method), 79
- clear() (kombu.async.timer.Timer method), 80
- clear() (kombu.simple.SimpleBuffer method), 53
- clear() (kombu.simple.SimpleQueue method), 52
- clear() (kombu.transport.virtual.BrokerState method), 156
- client (kombu.transport.base.Transport attribute), 151
- client (kombu.transport.beanstalk.Channel attribute), 139
- client (kombu.transport.beanstalk.Transport.Channel attribute), 138
- client (kombu.transport.couchdb.Channel attribute), 141
- client (kombu.transport.couchdb.Transport.Channel attribute), 140
- client (kombu.transport.mongodb.Channel attribute), 140
- client (kombu.transport.mongodb.Transport.Channel attribute), 139
- client (kombu.transport.redis.Channel attribute), 137
- client (kombu.transport.redis.Transport.Channel attribute), 135
- client (kombu.transport.zookeeper.Channel attribute), 142
- client (kombu.transport.zookeeper.Transport.Channel attribute), 142
- client\_heartbeat (kombu.transport.pyamqp.Connection attribute), 98
- clock (kombu.clocks.timetuple attribute), 54
- clone() (kombu.Connection method), 35
- clone() (kombu.connection.Connection method), 67
- close() (kombu.async.Hub method), 76
- close() (kombu.async.hub.Hub method), 77
- close() (kombu.compat.Consumer method), 57
- close() (kombu.compat.ConsumerSet method), 59
- close() (kombu.compat.Publisher method), 55
- close() (kombu.Connection method), 36
- close() (kombu.connection.Connection method), 67
- close() (kombu.pools.ProducerPool.Producer method), 74
- close() (kombu.simple.SimpleBuffer method), 53
- close() (kombu.simple.SimpleQueue method), 52
- close() (kombu.transport.amqplib.Channel method), 149
- close() (kombu.transport.beanstalk.Channel method), 139
- close() (kombu.transport.beanstalk.Transport.Channel method), 138
- close() (kombu.transport.memory.Channel method), 134
- close() (kombu.transport.memory.Transport.Channel method), 134
- close() (kombu.transport.pyamqp.Connection method), 98
- close() (kombu.transport.pyamqp.Connection.Channel method), 89
- close() (kombu.transport.qpid.Channel method), 129
- close() (kombu.transport.qpid.Connection method), 124
- close() (kombu.transport.qpid.Connection.Channel method), 120
- close() (kombu.transport.qpid.Transport.Connection method), 111
- close() (kombu.transport.qpid.Transport.Connection.Channel method), 107
- close() (kombu.transport.redis.Channel method), 137
- close() (kombu.transport.redis.Transport.Channel method), 135
- close() (kombu.transport.virtual.Channel method), 154
- close() (kombu.transport.virtual.scheduling.FairCycle method), 157
- close\_channel() (kombu.transport.base.Transport method), 151
- close\_channel() (kombu.transport.qpid.Connection method), 124
- close\_channel() (kombu.transport.qpid.Transport.Connection method), 111
- close\_channel() (kombu.transport.virtual.Transport method), 152
- close\_connection() (kombu.transport.amqplib.Transport method), 148
- close\_connection() (kombu.transport.base.Transport method), 151
- close\_connection() (kombu.transport.pyamqp.Transport method), 82
- close\_connection() (kombu.transport.qpid.Transport method), 112
- close\_connection() (kombu.transport.virtual.Transport method), 152
- close\_resource() (kombu.pools.ProducerPool method), 75
- codecs (kombu.transport.qpid.Channel attribute), 129
- codecs (kombu.transport.qpid.Connection.Channel attribute), 120
- codecs (kombu.transport.qpid.Transport.Connection.Channel attribute), 107
- collect() (kombu.connection.Connection method), 67
- collect\_replies() (in module kombu.common), 49
- commit\_on\_success() (in module kombu.transport.django.managers), 145
- completes\_cycle() (kombu.Connection method), 36
- completes\_cycle() (kombu.connection.Connection method), 67
- compress() (in module kombu.compression), 72
- compression (kombu.compat.Publisher attribute), 55
- compression (kombu.pools.ProducerPool.Producer

- attribute), 74
- compression (kombu.Producer attribute), 44
- confirm\_select() (kombu.transport.pyamqp.Connection.Channel method), 89
- conn\_or\_acquire() (kombu.transport.redis.Channel method), 137
- conn\_or\_acquire() (kombu.transport.redis.Transport.Channel method), 135
- connect() (kombu.Connection method), 34
- connect() (kombu.connection.Connection method), 67
- connect\_max\_retries (kombu.mixins.ConsumerMixin attribute), 51
- connect\_timeout (kombu.Connection attribute), 33
- connect\_timeout (kombu.connection.Connection attribute), 67
- connected (kombu.Connection attribute), 33
- connected (kombu.connection.Connection attribute), 67
- connected (kombu.transport.amqplib.Connection attribute), 149
- connected (kombu.transport.amqplib.Transport.Connection attribute), 148
- connected (kombu.transport.pyamqp.Connection attribute), 98
- Connection (class in kombu), 31
- Connection (class in kombu.connection), 64
- Connection (class in kombu.transport.amqplib), 149
- Connection (class in kombu.transport.pyamqp), 83
- Connection (class in kombu.transport.qpid), 114
- connection (kombu.compat.Consumer attribute), 57
- connection (kombu.compat.ConsumerSet attribute), 59
- connection (kombu.compat.Publisher attribute), 55
- connection (kombu.Connection attribute), 33
- connection (kombu.connection.Connection attribute), 67
- connection (kombu.Consumer attribute), 46
- connection (kombu.pidbox.Mailbox attribute), 62
- connection (kombu.pools.ProducerPool.Producer attribute), 74
- connection (kombu.Producer attribute), 44
- Connection (kombu.transport.amqplib.Connection attribute), 149
- Connection.Channel (class in kombu.transport.pyamqp), 83
- Connection.Channel (class in kombu.transport.qpid), 115
- Connection.Channel.Message (class in kombu.transport.pyamqp), 83
- Connection.Channel.Message (class in kombu.transport.qpid), 116
- Connection.Channel.Message.MessageStateError, 83
- Connection.Channel.QoS (class in kombu.transport.qpid), 116
- Connection.SSLTransport (class in kombu.transport.amqplib), 149
- Connection.TCPTransport (class in kombu.transport.amqplib), 149
- connection\_errors (kombu.Connection attribute), 33
- connection\_errors (kombu.connection.Connection attribute), 67
- connection\_errors (kombu.mixins.ConsumerMixin attribute), 51
- connection\_errors (kombu.transport.amqplib.Transport attribute), 148
- connection\_errors (kombu.transport.base.Transport attribute), 151
- connection\_errors (kombu.transport.beanstalk.Transport attribute), 138
- connection\_errors (kombu.transport.couchdb.Transport attribute), 141
- connection\_errors (kombu.transport.mongodb.Transport attribute), 140
- connection\_errors (kombu.transport.pyamqp.Connection attribute), 98
- connection\_errors (kombu.transport.pyamqp.Transport attribute), 82
- connection\_errors (kombu.transport.qpid.Transport attribute), 112
- connection\_errors (kombu.transport.SLMQ.Transport attribute), 146
- connection\_errors (kombu.transport.zookeeper.Transport attribute), 142
- connection\_for\_write() (kombu.transport.django.managers.MessageManager method), 145
- ConnectionLimitExceeded, 63
- ConnectionPool (class in kombu.connection), 70
- conninfo (kombu.transport.SLMQ.Channel attribute), 147
- conninfo (kombu.transport.SLMQ.Transport.Channel attribute), 146
- consume() (kombu.compat.Consumer method), 57
- consume() (kombu.compat.ConsumerSet method), 59
- consume() (kombu.Consumer method), 47
- consume() (kombu.mixins.ConsumerMixin method), 51
- consume() (kombu.Queue method), 42
- Consumer (class in kombu), 45
- Consumer (class in kombu.compat), 56
- consumer (kombu.simple.SimpleBuffer attribute), 53
- consumer (kombu.simple.SimpleQueue attribute), 52
- Consumer() (kombu.Connection method), 36
- Consumer() (kombu.connection.Connection method), 66
- Consumer() (kombu.mixins.ConsumerMixin method), 51
- Consumer() (kombu.pidbox.Node method), 63
- Consumer() (kombu.transport.pyamqp.Connection.Channel method), 83
- Consumer.ContentDisallowed, 56
- consumer\_context() (kombu.mixins.ConsumerMixin method), 51
- ConsumerMixin (class in kombu.mixins), 50
- ConsumerSet (class in kombu.compat), 59
- ConsumerSet.ContentDisallowed, 59



- consuming\_from() (kombu.compat.Consumer method), 57  
 consuming\_from() (kombu.compat.ConsumerSet method), 60  
 consuming\_from() (kombu.Consumer method), 47  
 content\_encoding (kombu.message.Message attribute), 72  
 content\_encoding (kombu.transport.base.Message attribute), 150  
 content\_encoding (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83  
 content\_encoding (kombu.transport.virtual.Message attribute), 154  
 content\_type (kombu.message.Message attribute), 72  
 content\_type (kombu.transport.base.Message attribute), 150  
 content\_type (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83  
 content\_type (kombu.transport.virtual.Message attribute), 154  
 create() (kombu.pools.PoolGroup method), 75  
 create\_broadcast\_cursor() (kombu.transport.mongodb.Channel method), 140  
 create\_broadcast\_cursor() (kombu.transport.mongodb.Transport.Channel method), 139  
 create\_channel() (kombu.transport.amqplib.Transport method), 148  
 create\_channel() (kombu.transport.base.Transport method), 151  
 create\_channel() (kombu.transport.pyamqp.Transport method), 82  
 create\_channel() (kombu.transport.qpid.Transport method), 112  
 create\_channel() (kombu.transport.virtual.Transport method), 152  
 create\_connection() (kombu.mixins.ConsumerMixin method), 51  
 create\_loop() (kombu.async.Hub method), 76  
 create\_loop() (kombu.async.hub.Hub method), 77  
 create\_message\_view() (in module kombu.transport.couchdb), 141  
 create\_producer() (kombu.pools.ProducerPool method), 75  
 create\_transport() (kombu.Connection method), 35  
 create\_transport() (kombu.connection.Connection method), 68  
 critical() (kombu.log.LogMixin method), 63  
 cycle (kombu.Connection attribute), 33  
 cycle (kombu.connection.Connection attribute), 68  
 Cycle (kombu.transport.virtual.Transport attribute), 152  
 cycle (kombu.transport.virtual.Transport attribute), 152
- ## D
- data\_folder\_in (kombu.transport.filesystem.Channel attribute), 143  
 data\_folder\_in (kombu.transport.filesystem.Transport.Channel attribute), 142  
 data\_folder\_out (kombu.transport.filesystem.Channel attribute), 143  
 data\_folder\_out (kombu.transport.filesystem.Transport.Channel attribute), 142  
 debug() (kombu.log.LogMixin method), 64  
 declare() (kombu.compat.Consumer method), 57  
 declare() (kombu.compat.ConsumerSet method), 60  
 declare() (kombu.compat.Publisher method), 55  
 declare() (kombu.Consumer method), 46  
 declare() (kombu.Exchange method), 40  
 declare() (kombu.pools.ProducerPool.Producer method), 74  
 declare() (kombu.Producer method), 45  
 declare() (kombu.Queue method), 43  
 declared\_entities (kombu.Connection attribute), 33  
 declared\_entities (kombu.connection.Connection attribute), 68  
 decode() (in module kombu.serialization), 158  
 decode() (kombu.message.Message method), 72  
 decode() (kombu.transport.base.Message method), 151  
 decode() (kombu.transport.pyamqp.Connection.Channel.Message method), 83  
 decode() (kombu.transport.virtual.Message method), 154  
 decode\_body() (kombu.transport.qpid.Channel method), 129  
 decode\_body() (kombu.transport.qpid.Connection.Channel method), 120  
 decode\_body() (kombu.transport.qpid.Transport.Connection.Channel method), 107  
 decompress() (in module kombu.compression), 73  
 default\_channel (kombu.Connection attribute), 33  
 default\_channel (kombu.connection.Connection attribute), 68  
 default\_connection\_params (kombu.transport.amqplib.Transport attribute), 148  
 default\_connection\_params (kombu.transport.pyamqp.Transport attribute), 82  
 default\_connection\_params (kombu.transport.qpid.Transport attribute), 112  
 default\_encode() (in module kombu.utils.encoding), 163  
 default\_encoding() (in module kombu.utils.encoding), 163  
 default\_encoding\_file (in module kombu.utils.encoding), 163  
 default\_port (kombu.transport.amqplib.Transport attribute), 148

- default\_port (kombu.transport.base.Transport attribute), 151
- default\_port (kombu.transport.beanstalk.Transport attribute), 138
- default\_port (kombu.transport.couchdb.Transport attribute), 141
- default\_port (kombu.transport.django.Transport attribute), 143
- default\_port (kombu.transport.filesystem.Transport attribute), 143
- default\_port (kombu.transport.mongodb.Transport attribute), 140
- default\_port (kombu.transport.pyamqp.Transport attribute), 82
- default\_port (kombu.transport.pyro.Transport attribute), 147
- default\_port (kombu.transport.redis.Transport attribute), 136
- default\_port (kombu.transport.SLMQ.Transport attribute), 146
- default\_port (kombu.transport.virtual.Transport attribute), 152
- default\_port (kombu.transport.zookeeper.Transport attribute), 142
- default\_ssl\_port (kombu.transport.pyamqp.Transport attribute), 82
- DEFAULT\_TRANSPORT (in module kombu.transport), 81
- default\_visibility\_timeout (kombu.transport.SLMQ.Channel attribute), 147
- default\_visibility\_timeout (kombu.transport.SLMQ.Transport.Channel attribute), 146
- delete() (kombu.Exchange method), 40
- delete() (kombu.Queue method), 43
- delete\_message() (kombu.transport.SLMQ.Channel method), 147
- delete\_message() (kombu.transport.SLMQ.Transport.Channel method), 146
- deleter() (kombu.utils.cached\_property method), 160
- deliver() (kombu.transport.virtual.exchange.DirectExchange method), 156
- deliver() (kombu.transport.virtual.exchange.FanoutExchange method), 157
- deliver() (kombu.transport.virtual.exchange.TopicExchange method), 156
- delivery\_info (kombu.message.Message attribute), 72
- delivery\_info (kombu.transport.base.Message attribute), 150
- delivery\_info (kombu.transport.pyamqp.Connection.Channel.Message attribute), 153
- delivery\_info (kombu.transport.virtual.Message attribute), 154
- delivery\_mode (kombu.Exchange attribute), 39, 40
- delivery\_tag (kombu.message.Message attribute), 72
- delivery\_tag (kombu.transport.base.Message attribute), 150
- delivery\_tag (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83
- delivery\_tag (kombu.transport.virtual.Message attribute), 154
- detect\_environment() (in module kombu.syn), 76
- DirectExchange (class in kombu.transport.virtual.exchange), 156
- disable\_insecure\_serializers() (in module kombu), 31
- discard\_all() (kombu.compat.Consumer method), 57
- discard\_all() (kombu.compat.ConsumerSet method), 60
- dispatch() (kombu.pidbox.Node method), 63
- dispatch\_from\_message() (kombu.pidbox.Node method), 63
- dispatch\_method() (kombu.transport.pyamqp.Connection method), 98
- dispatch\_method() (kombu.transport.pyamqp.Connection.Channel method), 89
- do\_restore (kombu.transport.memory.Channel attribute), 134
- do\_restore (kombu.transport.memory.Transport.Channel attribute), 134
- do\_restore (kombu.transport.virtual.Channel attribute), 152
- domain\_format (kombu.transport.SLMQ.Channel attribute), 147
- domain\_format (kombu.transport.SLMQ.Transport.Channel attribute), 146
- drain\_consumer() (in module kombu.common), 49
- drain\_events() (kombu.Connection method), 34
- drain\_events() (kombu.connection.Connection method), 68
- drain\_events() (kombu.transport.amqplib.Connection method), 149
- drain\_events() (kombu.transport.amqplib.Transport method), 148
- drain\_events() (kombu.transport.amqplib.Transport.Connection method), 148
- drain\_events() (kombu.transport.base.Transport method), 151
- drain\_events() (kombu.transport.pyamqp.Connection method), 98
- drain\_events() (kombu.transport.pyamqp.Transport method), 82
- drain\_events() (kombu.transport.qpid.Transport method), 112
- drain\_events() (kombu.transport.virtual.Channel method), 152
- drain\_events() (kombu.transport.virtual.Transport method), 152
- driver\_name (kombu.transport.amqplib.Transport at-

- tribute), 148
  - driver\_name (kombu.transport.beanstalk.Transport attribute), 138
  - driver\_name (kombu.transport.couchdb.Transport attribute), 141
  - driver\_name (kombu.transport.django.Transport attribute), 143
  - driver\_name (kombu.transport.filesystem.Transport attribute), 143
  - driver\_name (kombu.transport.memory.Transport attribute), 134
  - driver\_name (kombu.transport.mongodb.Transport attribute), 140
  - driver\_name (kombu.transport.pyamqp.Transport attribute), 82
  - driver\_name (kombu.transport.pyro.Transport attribute), 147
  - driver\_name (kombu.transport.qpid.Transport attribute), 112
  - driver\_name (kombu.transport.redis.Transport attribute), 136
  - driver\_name (kombu.transport.zookeeper.Transport attribute), 142
  - driver\_type (kombu.transport.amqplib.Transport attribute), 148
  - driver\_type (kombu.transport.beanstalk.Transport attribute), 139
  - driver\_type (kombu.transport.couchdb.Transport attribute), 141
  - driver\_type (kombu.transport.django.Transport attribute), 143
  - driver\_type (kombu.transport.filesystem.Transport attribute), 143
  - driver\_type (kombu.transport.memory.Transport attribute), 134
  - driver\_type (kombu.transport.mongodb.Transport attribute), 140
  - driver\_type (kombu.transport.pyamqp.Transport attribute), 82
  - driver\_type (kombu.transport.pyro.Transport attribute), 147
  - driver\_type (kombu.transport.qpid.Transport attribute), 112
  - driver\_type (kombu.transport.redis.Transport attribute), 136
  - driver\_type (kombu.transport.zookeeper.Transport attribute), 142
  - driver\_version() (kombu.transport.beanstalk.Transport method), 139
  - driver\_version() (kombu.transport.couchdb.Transport method), 141
  - driver\_version() (kombu.transport.django.Transport method), 143
  - driver\_version() (kombu.transport.filesystem.Transport method), 143
  - driver\_version() (kombu.transport.memory.Transport method), 134
  - driver\_version() (kombu.transport.mongodb.Transport method), 140
  - driver\_version() (kombu.transport.pyamqp.Transport method), 82
  - driver\_version() (kombu.transport.pyro.Transport method), 147
  - driver\_version() (kombu.transport.redis.Transport method), 136
  - driver\_version() (kombu.transport.zookeeper.Transport method), 142
  - DummyLock (class in kombu.async.semaphore), 78
  - dumps() (in module kombu.serialization), 159
  - durable (kombu.compat.Consumer attribute), 57
  - durable (kombu.compat.Publisher attribute), 55
  - durable (kombu.Exchange attribute), 39, 40
  - durable (kombu.Queue attribute), 42, 43
- ## E
- emergency\_dump\_state() (in module kombu.utils), 160
  - enable\_insecure\_serializers() (in module kombu), 31
  - encode() (in module kombu.serialization), 158
  - encode\_body() (kombu.transport.qpid.Channel method), 129
  - encode\_body() (kombu.transport.qpid.Connection.Channel method), 120
  - encode\_body() (kombu.transport.qpid.Transport.Connection.Channel method), 107
  - encoders() (in module kombu.compression), 73
  - ensure() (kombu.Connection method), 35
  - ensure() (kombu.connection.Connection method), 68
  - ensure\_bytes() (in module kombu.utils.encoding), 163
  - ensure\_connection() (kombu.Connection method), 34
  - ensure\_connection() (kombu.connection.Connection method), 68
  - enter\_after() (kombu.async.timer.Timer method), 80
  - enter\_at() (kombu.async.timer.Timer method), 80
  - entity\_name() (kombu.transport.SLMQ.Channel method), 147
  - entity\_name() (kombu.transport.SLMQ.Transport.Channel method), 146
  - Entry (class in kombu.async.timer), 79
  - environment variable
    - AWS\_ACCESS\_KEY\_ID, 209
    - AWS\_SECRET\_ACCESS\_KEY, 209
    - KOMBU\_LOG\_CHANNEL, 212
    - KOMBU\_LOG\_CONNECTION, 211
    - KOMBU\_LOG\_DEBUG, 211, 212
    - PICKLE\_PROTOCOL, 26, 191
    - URL, 195
    - VHOST, 195
  - EqualityDict (class in kombu.utils), 159

- equivalent() (kombu.transport.virtual.exchange.ExchangeType attribute), 157
  - ERR (kombu.async.Hub attribute), 76
  - ERR (kombu.async.hub.Hub attribute), 77
  - error() (kombu.log.LogMixin method), 64
  - errors (kombu.message.Message attribute), 72
  - errors (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83
  - errors (kombu.transport.virtual.Message attribute), 154
  - establish\_connection() (kombu.mixins.ConsumerMixin method), 51
  - establish\_connection() (kombu.transport.amqplib.Transport method), 149
  - establish\_connection() (kombu.transport.base.Transport method), 151
  - establish\_connection() (kombu.transport.pyamqp.Transport method), 82
  - establish\_connection() (kombu.transport.qpid.Transport method), 112
  - establish\_connection() (kombu.transport.virtual.Transport method), 152
  - evaluate() (kombu.utils.functional.lazy method), 163
  - eventloop() (in module kombu.common), 49
  - events (kombu.transport.amqplib.Channel attribute), 149
  - Exchange (class in kombu), 38
  - exchange (kombu.compat.Consumer attribute), 57
  - exchange (kombu.compat.Publisher attribute), 55
  - exchange (kombu.pidbox.Mailbox attribute), 62
  - exchange (kombu.pools.ProducerPool.Producer attribute), 74
  - exchange (kombu.Producer attribute), 44
  - exchange (kombu.Queue attribute), 41, 43
  - exchange\_bind() (kombu.transport.pyamqp.Connection.Channel method), 89
  - exchange\_declare() (kombu.transport.pyamqp.Connection.Channel method), 90
  - exchange\_declare() (kombu.transport.qpid.Channel method), 130
  - exchange\_declare() (kombu.transport.qpid.Connection.Channel method), 120
  - exchange\_declare() (kombu.transport.qpid.Transport.Connection.Channel method), 107
  - exchange\_declare() (kombu.transport.virtual.Channel method), 153
  - exchange\_delete() (kombu.transport.pyamqp.Connection.Channel method), 91
  - exchange\_delete() (kombu.transport.qpid.Channel method), 130
  - exchange\_delete() (kombu.transport.qpid.Connection.Channel method), 121
  - exchange\_delete() (kombu.transport.qpid.Transport.Connection.Channel method), 108
  - exchange\_delete() (kombu.transport.virtual.Channel method), 153
  - exchange\_opts (kombu.simple.SimpleBuffer attribute), 53
  - exchange\_opts (kombu.simple.SimpleQueue attribute), 52
  - exchange\_type (kombu.compat.Consumer attribute), 57
  - exchange\_type (kombu.compat.Publisher attribute), 55
  - exchange\_types (kombu.transport.virtual.Channel attribute), 153
  - exchange\_unbind() (kombu.transport.pyamqp.Connection.Channel method), 92
  - exchanges (kombu.transport.virtual.BrokerState attribute), 156
  - ExchangeType (class in kombu.transport.virtual.exchange), 157
  - exclusive (kombu.compat.Consumer attribute), 57
  - exclusive (kombu.Queue attribute), 42, 43
  - exec\_() (in module kombu.five), 164
  - expected\_time() (kombu.utils.limits.TokenBucket method), 161
  - extra\_context() (kombu.mixins.ConsumerMixin method), 51
- ## F
- failover\_strategies (kombu.connection.Connection attribute), 69
  - failover\_strategy (kombu.Connection attribute), 33
  - failover\_strategy (kombu.connection.Connection attribute), 69
  - FairCycle (class in kombu.transport.virtual.scheduling), 157
  - fanout\_patterns (kombu.transport.redis.Channel attribute), 137
  - fanout\_patterns (kombu.transport.redis.Transport.Channel attribute), 135
  - fanout\_prefix (kombu.transport.redis.Channel attribute), 137
  - fanout\_prefix (kombu.transport.redis.Transport.Channel attribute), 135
  - FanoutExchange (class in kombu.transport.virtual.exchange), 157
  - fanout(kombu.compat.Consumer method), 57
  - fetch() (kombu.transport.django.managers.QueueManager method), 145
  - fileno() (in module kombu.utils), 161
  - finalize (kombu.utils.limits.TokenBucket attribute), 162
  - fire\_timers() (kombu.async.Hub method), 76
  - fire\_timers() (kombu.async.hub.Hub method), 78
  - flow() (kombu.compat.Consumer method), 57
  - flow() (kombu.compat.ConsumerSet method), 60
  - flow() (kombu.Consumer method), 47
  - flow() (kombu.transport.pyamqp.Connection.Channel method), 92
  - flow() (kombu.transport.virtual.Channel method), 153
  - fmatch\_best() (in module kombu.utils.text), 164

- fmatch\_iter() (in module kombu.utils.text), 164  
 force\_close\_all() (kombu.connection.ChannelPool method), 71  
 force\_close\_all() (kombu.connection.ConnectionPool method), 71  
 forward() (kombu.clocks.LamportClock method), 54  
 from\_dict() (kombu.Queue class method), 43  
 from\_transport\_options (kombu.transport.redis.Channel attribute), 137  
 from\_transport\_options (kombu.transport.redis.Transport.Channel attribute), 135  
 from\_utf8() (in module kombu.utils.encoding), 163  
 fun (kombu.async.timer.Entry attribute), 79  
 fun (kombu.async.timer.Timer.Entry attribute), 80  
 fxrange() (in module kombu.utils), 160  
 fxrangelimit() (in module kombu.utils), 160
- ## G
- get() (kombu.Queue method), 43  
 get() (kombu.simple.SimpleBuffer method), 53  
 get() (kombu.simple.SimpleQueue method), 52  
 get() (kombu.transport.qpid.Channel.QoS method), 126  
 get() (kombu.transport.qpid.Connection.Channel.QoS method), 117  
 get() (kombu.transport.qpid.Transport.Connection.Channel.QoS method), 104  
 get() (kombu.transport.virtual.QoS method), 155  
 get() (kombu.transport.virtual.scheduling.FairCycle method), 157  
 get\_bindings() (kombu.transport.pyamqp.Connection.Channel method), 93  
 get\_broadcast() (kombu.transport.mongodb.Channel method), 140  
 get\_broadcast() (kombu.transport.mongodb.Transport.Channel method), 139  
 get\_broadcast\_cursor() (kombu.transport.mongodb.Channel method), 140  
 get\_broadcast\_cursor() (kombu.transport.mongodb.Transport.Channel method), 139  
 get\_consumers() (kombu.mixins.ConsumerMixin method), 51  
 get\_decoder() (in module kombu.compression), 73  
 get\_default\_encoding\_file() (in module kombu.utils.encoding), 163  
 get\_encoder() (in module kombu.compression), 73  
 get\_errno() (in module kombu.utils.compat), 162  
 get\_event\_loop() (in module kombu.async), 77  
 get\_event\_loop() (in module kombu.async.hub), 78  
 get\_heartbeat\_interval() (kombu.connection.Connection method), 69  
 get\_heartbeat\_interval() (kombu.transport.pyamqp.Transport.Channel method), 82  
 get\_limit() (in module kombu.pools), 75  
 get\_logger() (kombu.log.LogMixin method), 64  
 get\_loglevel() (in module kombu.log), 64  
 get\_loglevel() (kombu.log.LogMixin method), 64  
 get\_manager() (in module kombu.utils.amqp\_manager), 164  
 get\_manager() (kombu.Connection method), 36  
 get\_manager() (kombu.connection.Connection method), 69  
 get\_manager() (kombu.transport.amqplib.Transport method), 149  
 get\_manager() (kombu.transport.pyamqp.Transport method), 82  
 get\_messages() (kombu.transport.mongodb.Channel method), 140  
 get\_messages() (kombu.transport.mongodb.Transport.Channel method), 139  
 get\_nowait() (kombu.simple.SimpleBuffer method), 53  
 get\_nowait() (kombu.simple.SimpleQueue method), 52  
 get\_qpid\_connection() (kombu.transport.qpid.Connection method), 124  
 get\_qpid\_connection() (kombu.transport.qpid.Transport.Connection method), 111  
 get\_queue() (kombu.pidbox.Mailbox method), 62  
 get\_reply\_queue() (kombu.pidbox.Mailbox method), 62  
 get\_routing() (kombu.transport.mongodb.Channel method), 140  
 get\_routing() (kombu.transport.mongodb.Transport.Channel method), 139  
 get\_table() (kombu.transport.mongodb.Channel method), 140  
 get\_table() (kombu.transport.mongodb.Transport.Channel method), 139  
 get\_table() (kombu.transport.redis.Channel method), 137  
 get\_table() (kombu.transport.redis.Transport.Channel method), 135  
 get\_table() (kombu.transport.virtual.Channel method), 153  
 get\_transport\_cls() (in module kombu.transport), 81  
 get\_transport\_cls() (kombu.Connection method), 35  
 get\_transport\_cls() (kombu.connection.Connection method), 69  
 grow() (kombu.async.semaphore.LaxBoundedSemaphore method), 79
- ## H
- handle() (kombu.pidbox.Node method), 63  
 handle\_call() (kombu.pidbox.Node method), 63  
 handle\_cast() (kombu.pidbox.Node method), 63  
 handle\_error() (kombu.async.timer.Timer method), 80  
 handle\_message() (kombu.pidbox.Node method), 63  
 handler() (kombu.pidbox.Node method), 63  
 handlers (kombu.pidbox.Node attribute), 63  
 headers (kombu.message.Message attribute), 72  
 headers (kombu.transport.base.Message attribute), 150

- headers (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83
  - headers (kombu.transport.virtual.Message attribute), 154
  - heartbeat (kombu.Connection attribute), 33
  - heartbeat (kombu.connection.Connection attribute), 69
  - heartbeat (kombu.transport.pyamqp.Connection attribute), 99
  - heartbeat\_check() (kombu.Connection method), 36
  - heartbeat\_check() (kombu.connection.Connection method), 69
  - heartbeat\_check() (kombu.transport.pyamqp.Transport method), 82
  - heartbeat\_tick() (kombu.transport.pyamqp.Connection method), 99
  - host (kombu.Connection attribute), 34
  - host (kombu.connection.Connection attribute), 69
  - hostname (kombu.Connection attribute), 33
  - hostname (kombu.connection.Connection attribute), 69
  - hostname (kombu.pidbox.Node attribute), 62
  - Hub (class in kombu.async), 76
  - Hub (class in kombu.async.hub), 77
- I**
- id (kombu.clocks.timetuple attribute), 54
  - id (kombu.transport.django.models.Message attribute), 144
  - id (kombu.transport.django.models.Queue attribute), 145
  - info() (kombu.Connection method), 35
  - info() (kombu.connection.Connection method), 69
  - info() (kombu.log.LogMixin method), 64
  - insured() (in module kombu.common), 49
  - is\_alive() (kombu.transport.amqplib.Transport method), 149
  - is\_alive() (kombu.transport.pyamqp.Connection method), 99
  - is\_bound (kombu.abstract.MaybeChannelBound attribute), 75
  - is\_enabled\_for() (kombu.log.LogMixin method), 64
  - is\_evented (kombu.Connection attribute), 34
  - is\_evented (kombu.connection.Connection attribute), 69
  - is\_list() (in module kombu.utils.functional), 163
  - items() (in module kombu.five), 164
  - iterconsume() (kombu.compat.Consumer method), 58
  - iterconsume() (kombu.compat.ConsumerSet method), 60
  - itermessages() (in module kombu.common), 49
  - iterqueue() (kombu.compat.Consumer method), 58
- K**
- key\_to\_pattern() (kombu.transport.virtual.exchange.TopicExchange method), 156
  - keyprefix\_fanout (kombu.transport.redis.Channel attribute), 137
  - keyprefix\_fanout (kombu.transport.redis.Transport.Channel attribute), 135
  - keyprefix\_queue (kombu.transport.redis.Channel attribute), 137
  - keyprefix\_queue (kombu.transport.redis.Transport.Channel attribute), 135
  - keys() (in module kombu.five), 164
  - kombu (module), 31
  - kombu.abstract (module), 75
  - kombu.async (module), 76
  - kombu.async.debug (module), 80
  - kombu.async.hub (module), 77
  - kombu.async.semaphore (module), 78
  - kombu.async.timer (module), 79
  - kombu.clocks (module), 53
  - kombu.common (module), 48
  - kombu.compat (module), 55
  - kombu.compression (module), 72
  - kombu.connection (module), 64
  - kombu.exceptions (module), 63
  - kombu.five (module), 164
  - kombu.log (module), 63
  - kombu.message (module), 71
  - kombu.mixins (module), 50
  - kombu.pidbox (module), 61
  - kombu.pools (module), 73
  - kombu.serialization (module), 157
  - kombu.simple (module), 51
  - kombu.syn (module), 76
  - kombu.transport (module), 81
  - kombu.transport.amqplib (module), 148
  - kombu.transport.base (module), 150
  - kombu.transport.beanstalk (module), 138
  - kombu.transport.couchdb (module), 140
  - kombu.transport.django (module), 143
  - kombu.transport.django.management.commands.clean\_kombu\_messages (module), 145
  - kombu.transport.django.managers (module), 145
  - kombu.transport.django.models (module), 144
  - kombu.transport.filesystem (module), 142
  - kombu.transport.memory (module), 133
  - kombu.transport.mongodb (module), 139
  - kombu.transport.pyamqp (module), 81
  - kombu.transport.pyro (module), 147
  - kombu.transport.qpid (module), 99
  - kombu.transport.redis (module), 134
  - kombu.transport.SLMQ (module), 146
  - kombu.transport.virtual (module), 151
  - kombu.transport.virtual.exchange (module), 156
  - kombu.transport.virtual.scheduling (module), 157
  - kombu.transport.zmq (module), 138
  - kombu.transport.zookeeper (module), 141
  - kombu.utils (module), 159
  - kombu.utils.amq\_manager (module), 164
  - kombu.utils.compat (module), 162
  - kombu.utils.debug (module), 162

kombu.utils.encoding (module), 162  
 kombu.utils.eventio (module), 161  
 kombu.utils.functional (module), 163  
 kombu.utils.limits (module), 161  
 kombu.utils.text (module), 164  
 kombu.utils.url (module), 163  
 KOMBU\_LOG\_CHANNEL, 212  
 KOMBU\_LOG\_CONNECTION, 211  
 KOMBU\_LOG\_DEBUG, 211, 212  
 kwargs (kombu.async.timer.Entry attribute), 79  
 kwargs (kombu.async.timer.Timer.Entry attribute), 80  
 kwdict() (in module kombu.utils), 160

## L

LamportClock (class in kombu.clocks), 53  
 last\_heartbeat\_received (kombu.transport.pyamqp.Connection attribute), 99  
 last\_heartbeat\_sent (kombu.transport.pyamqp.Connection attribute), 99  
 LaxBoundedSemaphore (class in kombu.async.semaphore), 78  
 lazy (class in kombu.utils.functional), 163  
 LimitExceeded, 63  
 LimitExceeded (kombu.connection.ChannelPool attribute), 71  
 LimitExceeded (kombu.connection.ConnectionPool attribute), 70  
 listen() (kombu.pidbox.Node method), 63  
 log() (kombu.log.LogMixin method), 64  
 logger (kombu.log.LogMixin attribute), 64  
 logger\_name (kombu.log.LogMixin attribute), 64  
 login\_method (kombu.Connection attribute), 33  
 login\_method (kombu.connection.Connection attribute), 69  
 LogMixin (class in kombu.log), 63  
 Logwrapped (class in kombu.utils.debug), 162  
 lookup() (kombu.transport.virtual.exchange.DirectExchange method), 156  
 lookup() (kombu.transport.virtual.exchange.ExchangeType method), 157  
 lookup() (kombu.transport.virtual.exchange.FanoutExchange method), 157  
 lookup() (kombu.transport.virtual.exchange.TopicExchange method), 156  
 loop (kombu.async.Hub attribute), 76  
 loop (kombu.async.hub.Hub attribute), 78

## M

Mailbox (class in kombu.pidbox), 62  
 mailbox (kombu.pidbox.Node attribute), 63  
 manager (kombu.Connection attribute), 34  
 manager (kombu.connection.Connection attribute), 69  
 max\_connections (kombu.transport.redis.Channel attribute), 137  
 max\_connections (kombu.transport.redis.Transport.Channel attribute), 135  
 maybe\_bind() (kombu.abstract.MaybeChannelBound method), 75  
 maybe\_bind() (kombu.Exchange method), 39  
 maybe\_bind() (kombu.Queue method), 42  
 maybe\_close\_channel() (kombu.Connection method), 36  
 maybe\_close\_channel() (kombu.connection.Connection method), 69  
 maybe\_conn\_error() (kombu.mixins.ConsumerMixin method), 51  
 maybe\_declare() (in module kombu.common), 48  
 maybe\_declare() (kombu.compat.Publisher method), 55  
 maybe\_declare() (kombu.pools.ProducerPool.Producer method), 74  
 maybe\_declare() (kombu.Producer method), 45  
 maybe\_evaluate() (in module kombu.utils.functional), 163  
 maybe\_fileno() (in module kombu.utils), 161  
 maybe\_list() (in module kombu.utils), 160  
 maybe\_list() (in module kombu.utils.functional), 163  
 maybe\_sanitize\_url() (in module kombu.utils.url), 163  
 maybe\_switch\_next() (kombu.Connection method), 36  
 maybe\_switch\_next() (kombu.connection.Connection method), 70  
 MaybeChannelBound (class in kombu.abstract), 75  
 Message (class in kombu.message), 71  
 Message (class in kombu.transport.amqplib), 150  
 Message (class in kombu.transport.base), 150  
 Message (class in kombu.transport.django.models), 144  
 Message (class in kombu.transport.pyamqp), 99  
 Message (class in kombu.transport.qpid), 133  
 Message (class in kombu.transport.virtual), 154  
 message (kombu.compat.Consumer.ContentDisallowed attribute), 56  
 message (kombu.compat.ConsumerSet.ContentDisallowed attribute), 59  
 message (kombu.transport.pyamqp.Connection.Channel.Message.MessageState attribute), 83  
 Message (kombu.transport.virtual.Channel attribute), 152  
 message (kombu.transport.virtual.Message.MessageStateError attribute), 154  
 Message() (kombu.Exchange method), 39  
 Message.DoesNotExist, 144  
 Message.MessageStateError, 71, 154  
 Message.MultipleObjectsReturned, 144  
 message\_to\_python() (kombu.transport.amqplib.Channel method), 149  
 message\_to\_python() (kombu.transport.pyamqp.Channel method), 99  
 message\_to\_python() (kombu.transport.pyamqp.Connection.Channel method), 93  
 message\_to\_python() (kombu.transport.pyamqp.Transport.Connection.Channel method), 82

message\_to\_python() (kombu.transport.virtual.Channel method), 153  
 MessageManager (class in kombu.transport.django.managers), 145  
 messages (kombu.transport.django.models.Queue attribute), 145  
 MessageStateError, 63  
 monotonic() (in module kombu.five), 164  
 multi\_call() (kombu.pidbox.Mailbox method), 62

## N

name (kombu.Exchange attribute), 38, 40  
 name (kombu.Queue attribute), 41, 43  
 name (kombu.transport.django.models.Queue attribute), 145  
 namespace (kombu.pidbox.Mailbox attribute), 62  
 nested() (in module kombu.utils), 161  
 new() (kombu.pools.ProducerPool method), 75  
 nextfun() (in module kombu.five), 164  
 no\_ack (kombu.compat.Consumer attribute), 58  
 no\_ack (kombu.compat.ConsumerSet attribute), 60  
 no\_ack (kombu.Consumer attribute), 46  
 no\_ack (kombu.Queue attribute), 43  
 no\_ack (kombu.simple.SimpleBuffer attribute), 53  
 no\_ack (kombu.simple.SimpleQueue attribute), 52  
 no\_ack\_consumers (kombu.transport.pyamqp.Connection.Channel attribute), 93  
 Node (class in kombu.pidbox), 62  
 Node() (kombu.pidbox.Mailbox method), 62  
 NotBoundError, 63

## O

obj (kombu.clocks.timetuple attribute), 54  
 objects (kombu.transport.django.models.Message attribute), 144  
 objects (kombu.transport.django.models.Queue attribute), 145  
 on\_close (kombu.async.Hub attribute), 76  
 on\_close (kombu.async.hub.Hub attribute), 78  
 on\_connection\_error() (kombu.mixins.ConsumerMixin method), 51  
 on\_connection\_revived() (kombu.mixins.ConsumerMixin method), 51  
 on\_consume\_end() (kombu.mixins.ConsumerMixin method), 51  
 on\_consume\_ready() (kombu.mixins.ConsumerMixin method), 51  
 on\_declared (kombu.Queue attribute), 42  
 on\_decode\_error (kombu.compat.Consumer attribute), 58  
 on\_decode\_error (kombu.compat.ConsumerSet attribute), 60  
 on\_decode\_error (kombu.Consumer attribute), 46

on\_decode\_error() (kombu.mixins.ConsumerMixin method), 51  
 on\_error (kombu.async.timer.Timer attribute), 80  
 on\_iteration() (kombu.mixins.ConsumerMixin method), 51  
 on\_message (kombu.compat.Consumer attribute), 58  
 on\_message (kombu.compat.ConsumerSet attribute), 60  
 on\_message (kombu.Consumer attribute), 46  
 on\_readable() (kombu.transport.qpid.Transport method), 113  
 on\_readable() (kombu.transport.redis.Transport method), 136  
 on\_return (kombu.compat.Publisher attribute), 55  
 on\_return (kombu.pools.ProducerPool.Producer attribute), 74  
 on\_return (kombu.Producer attribute), 44

## P

parse\_url() (in module kombu.utils.url), 163  
 passive (kombu.Exchange attribute), 40  
 password (kombu.Connection attribute), 33  
 password (kombu.connection.Connection attribute), 70  
 payload (kombu.message.Message attribute), 72  
 payload (kombu.transport.base.Message attribute), 150  
 payload (kombu.transport.django.models.Message attribute), 144  
 payload (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83  
 payload (kombu.transport.virtual.Message attribute), 154  
 PERSISTENT\_DELIVERY\_MODE (kombu.Exchange attribute), 39  
 PICKLE\_PROTOCOL, 26, 191  
 pipe\_or\_acquire() (kombu.transport.redis.Channel.QoS method), 136  
 pipe\_or\_acquire() (kombu.transport.redis.Transport.Channel.QoS method), 135  
 poll() (in module kombu.utils.eventio), 161  
 polling\_interval (kombu.transport.beanstalk.Transport attribute), 139  
 polling\_interval (kombu.transport.couchdb.Transport attribute), 141  
 polling\_interval (kombu.transport.django.Transport attribute), 143  
 polling\_interval (kombu.transport.mongodb.Transport attribute), 140  
 polling\_interval (kombu.transport.qpid.Transport attribute), 113  
 polling\_interval (kombu.transport.redis.Transport attribute), 136  
 polling\_interval (kombu.transport.SLMQ.Transport attribute), 146  
 polling\_interval (kombu.transport.virtual.Transport attribute), 152



- polling\_interval (kombu.transport.zookeeper.Transport attribute), 142
- pool (kombu.transport.redis.Channel attribute), 137
- pool (kombu.transport.redis.Transport.Channel attribute), 135
- Pool() (kombu.Connection method), 36
- Pool() (kombu.connection.Connection method), 66
- PoolGroup (class in kombu.pools), 75
- pop() (kombu.transport.django.managers.MessageManager method), 145
- port (kombu.Connection attribute), 33
- port (kombu.connection.Connection attribute), 70
- prefetch\_count (kombu.transport.virtual.QoS attribute), 155
- prepare() (kombu.pools.ProducerPool method), 75
- prepare\_bind() (kombu.transport.virtual.exchange.ExchangeType method), 157
- prepare\_bind() (kombu.transport.virtual.exchange.TopicExchangeType method), 156
- prepare\_message() (kombu.transport.amqplib.Channel method), 149
- prepare\_message() (kombu.transport.pyamqp.Channel method), 99
- prepare\_message() (kombu.transport.pyamqp.Connection.Channel method), 93
- prepare\_message() (kombu.transport.pyamqp.Transport.Connection.Channel method), 82
- prepare\_message() (kombu.transport.qpid.Channel method), 130
- prepare\_message() (kombu.transport.qpid.Connection.Channel method), 121
- prepare\_message() (kombu.transport.qpid.Transport.Connection.Channel method), 108
- prepare\_message() (kombu.transport.virtual.Channel method), 153
- prev\_recv (kombu.transport.pyamqp.Connection attribute), 99
- prev\_sent (kombu.transport.pyamqp.Connection attribute), 99
- priority() (kombu.transport.redis.Channel method), 137
- priority() (kombu.transport.redis.Transport.Channel method), 136
- priority\_steps (kombu.transport.redis.Channel attribute), 137
- priority\_steps (kombu.transport.redis.Transport.Channel attribute), 136
- process\_next() (kombu.compat.Consumer method), 58
- processed\_folder (kombu.transport.filesystem.Channel attribute), 143
- processed\_folder (kombu.transport.filesystem.Transport.Channel attribute), 142
- Producer (class in kombu), 44
- producer (kombu.simple.SimpleBuffer attribute), 53
- producer (kombu.simple.SimpleQueue attribute), 52
- Producer() (kombu.Connection method), 36
- Producer() (kombu.connection.Connection method), 66
- Producer() (kombu.transport.pyamqp.Connection.Channel method), 84
- ProducerPool (class in kombu.pools), 73
- ProducerPool.Producer (class in kombu.pools), 73
- properties (kombu.message.Message attribute), 72
- properties (kombu.transport.base.Message attribute), 150
- properties (kombu.transport.pyamqp.Connection.Channel.Message attribute), 83
- properties (kombu.transport.virtual.Message attribute), 154
- publish() (kombu.compat.Publisher method), 55
- publish() (kombu.Exchange method), 40
- publish() (kombu.pools.ProducerPool.Producer method), 74
- publish() (kombu.Producer method), 45
- publish() (kombu.transport.django.managers.QueueManager method), 145
- Publisher (class in kombu.compat), 55
- purge() (kombu.compat.Consumer method), 58
- purge() (kombu.compat.ConsumerSet method), 60
- purge() (kombu.Consumer method), 47
- purge() (kombu.Queue method), 43
- purge() (kombu.transport.django.managers.QueueManager method), 145
- put() (kombu.simple.SimpleBuffer method), 53
- put() (kombu.simple.SimpleQueue method), 52

## Q

- QoS (class in kombu.transport.virtual), 155
- qos (kombu.transport.qpid.Channel attribute), 131
- qos (kombu.transport.qpid.Connection.Channel attribute), 121
- qos (kombu.transport.qpid.Transport.Connection.Channel attribute), 108
- qos (kombu.transport.virtual.Channel attribute), 152
- qos() (kombu.compat.Consumer method), 58
- qos() (kombu.compat.ConsumerSet method), 60
- qos() (kombu.Consumer method), 47
- qos\_semantics\_matches\_spec (kombu.connection.Connection attribute), 70
- qos\_semantics\_matches\_spec() (kombu.transport.pyamqp.Transport method), 82
- qsize() (kombu.simple.SimpleBuffer method), 53
- qsize() (kombu.simple.SimpleQueue method), 52
- Queue (class in kombu), 41
- Queue (class in kombu.transport.django.models), 144
- queue (kombu.async.timer.Timer attribute), 80
- queue (kombu.compat.Consumer attribute), 58
- queue (kombu.simple.SimpleBuffer attribute), 53
- queue (kombu.simple.SimpleQueue attribute), 52

- Queue (kombu.transport.django.Channel attribute), 144
  - queue (kombu.transport.django.models.Message attribute), 144
  - Queue (kombu.transport.django.Transport.Channel attribute), 143
  - Queue.ContentDisallowed, 42
  - Queue.DoesNotExist, 144
  - Queue.MultipleObjectsReturned, 144
  - queue\_arguments (kombu.Queue attribute), 42
  - queue\_bind() (kombu.Queue method), 43
  - queue\_bind() (kombu.transport.pyamqp.Connection.Channel method), 93
  - queue\_bind() (kombu.transport.qpid.Channel method), 131
  - queue\_bind() (kombu.transport.qpid.Connection.Channel method), 121
  - queue\_bind() (kombu.transport.qpid.Transport.Connection.Channel method), 109
  - queue\_bind() (kombu.transport.virtual.Channel method), 153
  - queue\_declare() (kombu.Queue method), 43
  - queue\_declare() (kombu.transport.pyamqp.Connection.Channel method), 94
  - queue\_declare() (kombu.transport.qpid.Channel method), 131
  - queue\_declare() (kombu.transport.qpid.Connection.Channel method), 122
  - queue\_declare() (kombu.transport.qpid.Transport.Connection.Channel method), 109
  - queue\_declare() (kombu.transport.virtual.Channel method), 153
  - queue\_delete() (kombu.transport.mongodb.Channel method), 140
  - queue\_delete() (kombu.transport.mongodb.Transport.Channel method), 139
  - queue\_delete() (kombu.transport.pyamqp.Connection.Channel method), 95
  - queue\_delete() (kombu.transport.qpid.Channel method), 132
  - queue\_delete() (kombu.transport.qpid.Connection.Channel method), 123
  - queue\_delete() (kombu.transport.qpid.Transport.Connection.Channel method), 110
  - queue\_delete() (kombu.transport.virtual.Channel method), 153
  - queue\_id (kombu.transport.django.models.Message attribute), 144
  - queue\_model (kombu.transport.django.Channel attribute), 144
  - queue\_model (kombu.transport.django.Transport.Channel attribute), 143
  - queue\_name\_prefix (kombu.transport.SLMQ.Channel attribute), 147
  - queue\_name\_prefix (kombu.transport.SLMQ.Transport.Channel attribute), 146
  - queue\_opts (kombu.simple.SimpleBuffer attribute), 53
  - queue\_opts (kombu.simple.SimpleQueue attribute), 52
  - queue\_purge() (kombu.transport.pyamqp.Connection.Channel method), 96
  - queue\_purge() (kombu.transport.qpid.Channel method), 132
  - queue\_purge() (kombu.transport.qpid.Connection.Channel method), 123
  - queue\_purge() (kombu.transport.qpid.Transport.Connection.Channel method), 110
  - queue\_purge() (kombu.transport.virtual.Channel method), 153
  - queue\_unbind() (kombu.Queue method), 44
  - queue\_unbind() (kombu.transport.pyamqp.Connection.Channel method), 97
  - queue\_unbind() (kombu.transport.qpid.Channel method), 133
  - queue\_unbind() (kombu.transport.qpid.Connection.Channel method), 123
  - queue\_unbind() (kombu.transport.qpid.Transport.Connection.Channel method), 111
  - QueueManager (class in kombu.transport.django.managers), 145
  - queues (kombu.compat.Consumer attribute), 58
  - queues (kombu.compat.ConsumerSet attribute), 60
  - queues (kombu.Consumer attribute), 46
  - queues (kombu.transport.memory.Channel attribute), 134
  - queues (kombu.transport.memory.Transport.Channel attribute), 134
  - queues() (kombu.transport.pyro.Channel method), 148
  - queues() (kombu.transport.pyro.Transport.Channel method), 147
- ## R
- raw\_encode() (in module kombu.serialization), 159
  - READ (kombu.async.Hub attribute), 76
  - READ (kombu.async.hub.Hub attribute), 77
  - read\_frame() (kombu.transport.amqplib.Connection.SSLTransport method), 149
  - read\_frame() (kombu.transport.amqplib.Connection.TCPTransport method), 149
  - read\_timeout() (kombu.transport.amqplib.Connection method), 149
  - read\_timeout() (kombu.transport.amqplib.Transport.Connection method), 148
  - read\_timeout() (kombu.transport.pyamqp.Connection method), 99
  - receive() (kombu.compat.Consumer method), 58
  - receive() (kombu.compat.ConsumerSet method), 60
  - receive() (kombu.Consumer method), 48
  - recover() (kombu.compat.Consumer method), 58
  - recover() (kombu.compat.ConsumerSet method), 61
  - recover() (kombu.Consumer method), 48

[recoverable\\_channel\\_errors](#) (kombu.Connection attribute), [33](#)  
[recoverable\\_channel\\_errors](#) (kombu.connection.Connection attribute), [70](#)  
[recoverable\\_channel\\_errors](#) (kombu.transport.base.Transport attribute), [151](#)  
[recoverable\\_channel\\_errors](#) (kombu.transport.pyamqp.Connection attribute), [99](#)  
[recoverable\\_channel\\_errors](#) (kombu.transport.pyamqp.Transport attribute), [82](#)  
[recoverable\\_channel\\_errors](#) (kombu.transport.qpid.Transport attribute), [113](#)  
[recoverable\\_connection\\_errors](#) (kombu.Connection attribute), [33](#)  
[recoverable\\_connection\\_errors](#) (kombu.connection.Connection attribute), [70](#)  
[recoverable\\_connection\\_errors](#) (kombu.transport.base.Transport attribute), [151](#)  
[recoverable\\_connection\\_errors](#) (kombu.transport.pyamqp.Connection attribute), [99](#)  
[recoverable\\_connection\\_errors](#) (kombu.transport.pyamqp.Transport attribute), [82](#)  
[recoverable\\_connection\\_errors](#) (kombu.transport.qpid.Transport attribute), [113](#)  
[refresh\\_connection\(\)](#) (kombu.transport.django.Channel method), [144](#)  
[refresh\\_connection\(\)](#) (kombu.transport.django.Transport.Channel method), [143](#)  
[register\(\)](#) (in module kombu.compression), [73](#)  
[register\(\)](#) (in module kombu.serialization), [159](#)  
[register\\_callback\(\)](#) (kombu.compat.Consumer method), [58](#)  
[register\\_callback\(\)](#) (kombu.compat.ConsumerSet method), [61](#)  
[register\\_callback\(\)](#) (kombu.Consumer method), [46](#)  
[register\\_group\(\)](#) (in module kombu.pools), [75](#)  
[register\\_with\\_event\\_loop\(\)](#) (kombu.Connection method), [36](#)  
[register\\_with\\_event\\_loop\(\)](#) (kombu.connection.Connection method), [70](#)  
[register\\_with\\_event\\_loop\(\)](#) (kombu.transport.amqplib.Transport method), [149](#)  
[register\\_with\\_event\\_loop\(\)](#) (kombu.transport.pyamqp.Transport method), [82](#)  
[register\\_with\\_event\\_loop\(\)](#) (kombu.transport.qpid.Transport method), [113](#)  
[register\\_with\\_event\\_loop\(\)](#) (kombu.transport.redis.Transport method), [136](#)  
[registry](#) (in module kombu.serialization), [159](#)  
[reject\(\)](#) (kombu.message.Message method), [72](#)  
[reject\(\)](#) (kombu.transport.base.Message method), [150](#)  
[reject\(\)](#) (kombu.transport.pyamqp.Connection.Channel.Message method), [83](#)  
[reject\(\)](#) (kombu.transport.qpid.Channel.QoS method), [126](#)  
[reject\(\)](#) (kombu.transport.qpid.Connection.Channel.QoS method), [117](#)  
[reject\(\)](#) (kombu.transport.qpid.Transport.Connection.Channel.QoS method), [104](#)  
[reject\(\)](#) (kombu.transport.redis.Channel.QoS method), [136](#)  
[reject\(\)](#) (kombu.transport.redis.Transport.Channel.QoS method), [135](#)  
[reject\(\)](#) (kombu.transport.virtual.Message method), [154](#)  
[reject\(\)](#) (kombu.transport.virtual.QoS method), [155](#)  
[reject\\_log\\_error\(\)](#) (kombu.message.Message method), [72](#)  
[reject\\_log\\_error\(\)](#) (kombu.transport.pyamqp.Connection.Channel.Message method), [84](#)  
[reject\\_log\\_error\(\)](#) (kombu.transport.virtual.Message method), [154](#)  
[release\(\)](#) (kombu.async.semaphore.LaxBoundedSemaphore method), [79](#)  
[release\(\)](#) (kombu.compat.Publisher method), [56](#)  
[release\(\)](#) (kombu.Connection method), [34](#)  
[release\(\)](#) (kombu.connection.ChannelPool method), [71](#)  
[release\(\)](#) (kombu.connection.Connection method), [70](#)  
[release\(\)](#) (kombu.connection.ConnectionPool method), [71](#)  
[release\(\)](#) (kombu.pools.ProducerPool method), [75](#)  
[release\(\)](#) (kombu.pools.ProducerPool.Producer method), [74](#)  
[remove\(\)](#) (kombu.async.Hub method), [76](#)  
[remove\(\)](#) (kombu.async.hub.Hub method), [78](#)  
[remove\\_reader\(\)](#) (kombu.async.Hub method), [76](#)  
[remove\\_reader\(\)](#) (kombu.async.hub.Hub method), [78](#)  
[remove\\_writer\(\)](#) (kombu.async.Hub method), [77](#)  
[remove\\_writer\(\)](#) (kombu.async.hub.Hub method), [78](#)  
[reply\(\)](#) (kombu.pidbox.Node method), [63](#)  
[reply\\_exchange](#) (kombu.pidbox.Mailbox attribute), [62](#)  
[repr\\_active\(\)](#) (in module kombu.async.debug), [80](#)  
[repr\\_active\(\)](#) (kombu.async.Hub method), [77](#)  
[repr\\_active\(\)](#) (kombu.async.hub.Hub method), [78](#)  
[repr\\_events\(\)](#) (in module kombu.async.debug), [80](#)

- repr\_events() (kombu.async.Hub method), 77
  - repr\_events() (kombu.async.hub.Hub method), 78
  - repr\_flag() (in module kombu.async.debug), 80
  - repr\_readers() (in module kombu.async.debug), 80
  - repr\_writers() (in module kombu.async.debug), 81
  - reprcall() (in module kombu.utils), 161
  - reprkwargs() (in module kombu.utils), 160
  - requeue() (kombu.message.Message method), 72
  - requeue() (kombu.transport.base.Message method), 151
  - requeue() (kombu.transport.pyamqp.Connection.Channel.Message method), 84
  - requeue() (kombu.transport.virtual.Message method), 154
  - reraise() (in module kombu.five), 164
  - reset() (in module kombu.pools), 75
  - reset() (kombu.async.Hub method), 77
  - reset() (kombu.async.hub.Hub method), 78
  - resolve\_transport() (in module kombu.transport), 81
  - restart\_limit (kombu.mixins.ConsumerMixin attribute), 51
  - restore\_at\_shutdown (kombu.transport.redis.Channel.QoS attribute), 136
  - restore\_at\_shutdown (kombu.transport.redis.Transport.Channel.QoS attribute), 135
  - restore\_at\_shutdown (kombu.transport.virtual.QoS attribute), 155
  - restore\_by\_tag() (kombu.transport.redis.Channel.QoS method), 136
  - restore\_by\_tag() (kombu.transport.redis.Transport.Channel.QoS method), 135
  - restore\_unacked() (kombu.transport.redis.Channel.QoS method), 137
  - restore\_unacked() (kombu.transport.redis.Transport.Channel.QoS method), 135
  - restore\_unacked() (kombu.transport.virtual.QoS method), 155
  - restore\_unacked\_once() (kombu.transport.virtual.QoS method), 155
  - restore\_visible() (kombu.transport.redis.Channel.QoS method), 137
  - restore\_visible() (kombu.transport.redis.Transport.Channel.QoS method), 135
  - restore\_visible() (kombu.transport.virtual.QoS method), 155
  - retry\_over\_time() (in module kombu.utils), 160
  - revive() (kombu.abstract.MaybeChannelBound method), 75
  - revive() (kombu.compat.Consumer method), 59
  - revive() (kombu.compat.ConsumerSet method), 61
  - revive() (kombu.compat.Publisher method), 56
  - revive() (kombu.Connection method), 35
  - revive() (kombu.connection.Connection method), 70
  - revive() (kombu.Consumer method), 48
  - revive() (kombu.pools.ProducerPool.Producer method), 74
  - revive() (kombu.Producer method), 45
  - routing\_key (kombu.compat.Consumer attribute), 59
  - routing\_key (kombu.compat.Publisher attribute), 56
  - routing\_key (kombu.pools.ProducerPool.Producer attribute), 75
  - routing\_key (kombu.Producer attribute), 44
  - routing\_key (kombu.Queue attribute), 41, 44
  - run() (kombu.mixins.ConsumerMixin method), 51
  - run\_forever() (kombu.async.Hub method), 77
  - run\_forever() (kombu.async.hub.Hub method), 78
  - run\_once() (kombu.async.Hub method), 77
  - run\_once() (kombu.async.hub.Hub method), 78
- ## S
- safe\_repr() (in module kombu.utils.encoding), 163
  - safe\_str() (in module kombu.utils.encoding), 163
  - sanitize\_url() (in module kombu.utils.url), 163
  - say() (in module kombu.utils), 159
  - schedule (kombu.async.timer.Timer attribute), 80
  - scheduler (kombu.async.Hub attribute), 77
  - scheduler (kombu.async.hub.Hub attribute), 78
  - select\_for\_update() (in module kombu.transport.django.managers), 145
  - send() (kombu.compat.Publisher method), 56
  - send\_heartbeat() (kombu.transport.pyamqp.Connection method), 99
  - send\_reply() (in module kombu.common), 49
  - sent\_at (kombu.transport.django.models.Message attribute), 144
  - sep (kombu.transport.redis.Channel attribute), 137
  - sep (kombu.transport.redis.Transport.Channel attribute), 136
  - serializable() (kombu.transport.qpid.Channel.Message method), 125
  - serializable() (kombu.transport.qpid.Connection.Channel.Message method), 116
  - serializable() (kombu.transport.qpid.Message method), 133
  - serializable() (kombu.transport.qpid.Transport.Connection.Channel.Message method), 103
  - serializable() (kombu.transport.virtual.Message method), 155
  - serializer (kombu.compat.Publisher attribute), 56
  - serializer (kombu.pools.ProducerPool.Producer attribute), 75
  - serializer (kombu.Producer attribute), 44
  - SerializerNotInstalled, 158
  - server\_capabilities (kombu.transport.pyamqp.Connection attribute), 99
  - server\_heartbeat (kombu.transport.pyamqp.Connection attribute), 99
  - set\_default\_encoding\_file() (in module kombu.utils.encoding), 163

- set\_event\_loop() (in module kombu.async), 77
  - set\_event\_loop() (in module kombu.async.hub), 78
  - set\_limit() (in module kombu.pools), 75
  - setter() (kombu.utils.cached\_property method), 160
  - setup() (kombu.pools.ProducerPool method), 75
  - setup\_logging() (in module kombu.log), 64
  - setup\_logging() (in module kombu.utils.debug), 162
  - shared\_queues (kombu.transport.pyro.Channel attribute), 148
  - shared\_queues (kombu.transport.pyro.Transport attribute), 147
  - shared\_queues (kombu.transport.pyro.Transport.Channel attribute), 147
  - should\_stop (kombu.mixins.ConsumerMixin attribute), 51
  - shrink() (kombu.async.semaphore.LaxBoundedSemaphore method), 79
  - SimpleBuffer (class in kombu.simple), 52
  - SimpleBuffer() (kombu.Connection method), 37
  - SimpleBuffer() (kombu.connection.Connection method), 66
  - SimpleQueue (class in kombu.simple), 52
  - SimpleQueue() (kombu.Connection method), 37
  - SimpleQueue() (kombu.connection.Connection method), 66
  - size() (kombu.transport.django.managers.QueueManager method), 145
  - slmq (kombu.transport.SLMQ.Channel attribute), 147
  - slmq (kombu.transport.SLMQ.Transport.Channel attribute), 146
  - sock (kombu.transport.pyamqp.Connection attribute), 99
  - socket\_connect\_timeout (kombu.transport.redis.Channel attribute), 137
  - socket\_connect\_timeout (kombu.transport.redis.Transport.Channel attribute), 136
  - socket\_keepalive (kombu.transport.redis.Channel attribute), 137
  - socket\_keepalive (kombu.transport.redis.Transport.Channel attribute), 136
  - socket\_keepalive\_options (kombu.transport.redis.Channel attribute), 137
  - socket\_keepalive\_options (kombu.transport.redis.Transport.Channel attribute), 136
  - socket\_timeout (kombu.transport.redis.Channel attribute), 137
  - socket\_timeout (kombu.transport.redis.Transport.Channel attribute), 136
  - sort\_heap() (kombu.clocks.LamportClock method), 54
  - ssl (kombu.Connection attribute), 33
  - ssl (kombu.connection.Connection attribute), 70
  - state (kombu.pidbox.Node attribute), 63
  - state (kombu.transport.memory.Transport attribute), 134
  - state (kombu.transport.pyro.Transport attribute), 147
  - state (kombu.transport.virtual.Channel attribute), 152
  - state (kombu.transport.virtual.Transport attribute), 152
  - stop() (kombu.async.Hub method), 77
  - stop() (kombu.async.hub.Hub method), 78
  - stop() (kombu.async.timer.Timer method), 80
  - store\_processed (kombu.transport.filesystem.Channel attribute), 143
  - store\_processed (kombu.transport.filesystem.Transport.Channel attribute), 142
  - str\_to\_bytes() (in module kombu.utils.encoding), 163
  - subclient (kombu.transport.redis.Channel attribute), 137
  - subclient (kombu.transport.redis.Transport.Channel attribute), 136
  - supports\_ev (kombu.transport.amqplib.Transport attribute), 149
  - supports\_ev (kombu.transport.pyamqp.Transport attribute), 82
  - supports\_ev (kombu.transport.qpid.Transport attribute), 114
  - supports\_ev (kombu.transport.redis.Transport attribute), 136
  - supports\_fanout (kombu.transport.memory.Channel attribute), 134
  - supports\_fanout (kombu.transport.memory.Transport.Channel attribute), 134
  - supports\_fanout (kombu.transport.mongodb.Channel attribute), 140
  - supports\_fanout (kombu.transport.mongodb.Transport.Channel attribute), 139
  - supports\_fanout (kombu.transport.redis.Channel attribute), 138
  - supports\_fanout (kombu.transport.redis.Transport.Channel attribute), 136
  - supports\_heartbeats (kombu.Connection attribute), 34
  - supports\_heartbeats (kombu.connection.Connection attribute), 70
  - supports\_heartbeats (kombu.transport.pyamqp.Transport attribute), 82
  - switch() (kombu.Connection method), 36
  - switch() (kombu.connection.Connection method), 70
- ## T
- timedelta\_seconds() (in module kombu.utils.compat), 162
  - TimeoutError (in module kombu.exceptions), 63
  - Timer (class in kombu.async.timer), 80
  - Timer.Entry (class in kombu.async.timer), 80
  - timespec (class in kombu.five), 164
  - timestamp (kombu.clocks.timetuple attribute), 54
  - timestamp (kombu.utils.limits.TokenBucket attribute), 162
  - timetuple (class in kombu.clocks), 54
  - to\_timestamp() (in module kombu.async.timer), 80
  - TokenBucket (class in kombu.utils.limits), 161

- TopicExchange (class in kombu.transport.virtual.exchange), 156
  - TRANSIENT\_DELIVERY\_MODE (kombu.Exchange attribute), 40
  - Transport (class in kombu.transport.amqpplib), 148
  - Transport (class in kombu.transport.base), 151
  - Transport (class in kombu.transport.beanstalk), 138
  - Transport (class in kombu.transport.couchdb), 140
  - Transport (class in kombu.transport.django), 143
  - Transport (class in kombu.transport.filesystem), 142
  - Transport (class in kombu.transport.memory), 134
  - Transport (class in kombu.transport.mongodb), 139
  - Transport (class in kombu.transport.pyamqp), 81
  - Transport (class in kombu.transport.pyro), 147
  - Transport (class in kombu.transport.qpid), 101
  - Transport (class in kombu.transport.redis), 135
  - Transport (class in kombu.transport.SLMQ), 146
  - Transport (class in kombu.transport.virtual), 152
  - Transport (class in kombu.transport.zookeeper), 142
  - transport (kombu.Connection attribute), 33
  - transport (kombu.connection.Connection attribute), 70
  - Transport() (kombu.transport.pyamqp.Connection method), 98
  - Transport.Channel (class in kombu.transport.beanstalk), 138
  - Transport.Channel (class in kombu.transport.couchdb), 140
  - Transport.Channel (class in kombu.transport.django), 143
  - Transport.Channel (class in kombu.transport.filesystem), 142
  - Transport.Channel (class in kombu.transport.memory), 134
  - Transport.Channel (class in kombu.transport.mongodb), 139
  - Transport.Channel (class in kombu.transport.pyro), 147
  - Transport.Channel (class in kombu.transport.redis), 135
  - Transport.Channel (class in kombu.transport.SLMQ), 146
  - Transport.Channel (class in kombu.transport.zookeeper), 142
  - Transport.Channel.QoS (class in kombu.transport.redis), 135
  - Transport.Connection (class in kombu.transport.amqpplib), 148
  - Transport.Connection (class in kombu.transport.pyamqp), 81
  - Transport.Connection (class in kombu.transport.qpid), 101
  - Transport.Connection.Channel (class in kombu.transport.pyamqp), 82
  - Transport.Connection.Channel (class in kombu.transport.qpid), 102
  - Transport.Connection.Channel.Message (class in kombu.transport.pyamqp), 82
  - Transport.Connection.Channel.Message (class in kombu.transport.qpid), 103
  - Transport.Connection.Channel.QoS (class in kombu.transport.qpid), 103
  - TRANSPORT\_ALIASES (in module kombu.transport), 81
  - transport\_options (kombu.connection.Connection attribute), 70
  - transport\_options (kombu.transport.filesystem.Channel attribute), 143
  - transport\_options (kombu.transport.filesystem.Transport.Channel attribute), 143
  - transport\_options (kombu.transport.SLMQ.Channel attribute), 147
  - transport\_options (kombu.transport.SLMQ.Transport.Channel attribute), 146
  - tref (kombu.async.timer.Entry attribute), 79
  - tref (kombu.async.timer.Timer.Entry attribute), 80
  - tv\_nsec (kombu.five.timespec attribute), 164
  - tv\_sec (kombu.five.timespec attribute), 164
  - tx\_commit() (kombu.transport.pyamqp.Connection.Channel method), 97
  - tx\_rollback() (kombu.transport.pyamqp.Connection.Channel method), 97
  - tx\_select() (kombu.transport.pyamqp.Connection.Channel method), 97
  - type (kombu.Exchange attribute), 38, 40
  - type (kombu.pidbox.Mailbox attribute), 62
  - type (kombu.transport.virtual.exchange.DirectExchange attribute), 156
  - type (kombu.transport.virtual.exchange.ExchangeType attribute), 157
  - type (kombu.transport.virtual.exchange.FanoutExchange attribute), 157
  - type (kombu.transport.virtual.exchange.TopicExchange attribute), 156
  - typeof() (kombu.transport.qpid.Channel method), 133
  - typeof() (kombu.transport.qpid.Connection.Channel method), 124
  - typeof() (kombu.transport.qpid.Transport.Connection.Channel method), 111
  - typeof() (kombu.transport.virtual.Channel method), 153
- ## U
- unacked\_index\_key (kombu.transport.redis.Channel attribute), 138
  - unacked\_index\_key (kombu.transport.redis.Channel.QoS attribute), 137
  - unacked\_index\_key (kombu.transport.redis.Transport.Channel attribute), 136
  - unacked\_index\_key (kombu.transport.redis.Transport.Channel.QoS attribute), 135
  - unacked\_key (kombu.transport.redis.Channel attribute), 138

- unacked\_key (kombu.transport.redis.Channel.QoS attribute), 137
- unacked\_key (kombu.transport.redis.Transport.Channel attribute), 136
- unacked\_key (kombu.transport.redis.Transport.Channel.QoS attribute), 135
- unacked\_mutex\_expire (kombu.transport.redis.Channel attribute), 138
- unacked\_mutex\_expire (kombu.transport.redis.Channel.QoS attribute), 137
- unacked\_mutex\_expire (kombu.transport.redis.Transport.Channel attribute), 136
- unacked\_mutex\_expire (kombu.transport.redis.Transport.Channel.QoS attribute), 135
- unacked\_mutex\_key (kombu.transport.redis.Channel attribute), 138
- unacked\_mutex\_key (kombu.transport.redis.Channel.QoS attribute), 137
- unacked\_mutex\_key (kombu.transport.redis.Transport.Channel attribute), 136
- unacked\_mutex\_key (kombu.transport.redis.Transport.Channel.QoS attribute), 135
- unacked\_restore\_limit (kombu.transport.redis.Channel attribute), 138
- unacked\_restore\_limit (kombu.transport.redis.Transport.Channel attribute), 136
- unbind\_from() (kombu.Exchange method), 40
- unbind\_from() (kombu.Queue method), 44
- uri\_prefix (kombu.Connection attribute), 33
- uri\_prefix (kombu.connection.Connection attribute), 70
- URL, 195
- userid (kombu.Connection attribute), 33
- userid (kombu.connection.Connection attribute), 70
- uuid() (in module kombu.common), 48
- uuid() (in module kombu.utils), 159

## V

- value (kombu.clocks.LamportClock attribute), 54
- values() (in module kombu.five), 164
- verify\_connection() (kombu.transport.amqplib.Transport method), 149
- verify\_connection() (kombu.transport.pyamqp.Transport method), 82
- verify\_runtime\_environment() (kombu.transport.qpid.Transport method), 114
- version\_string\_as\_tuple() (in module kombu.utils.text), 164
- VHOST, 195
- view\_created (kombu.transport.couchdb.Channel attribute), 141
- view\_created (kombu.transport.couchdb.Transport.Channel attribute), 141
- virtual\_host (kombu.Connection attribute), 33

- virtual\_host (kombu.connection.Connection attribute), 70
- visibility\_timeout (kombu.transport.redis.Channel attribute), 138
- visibility\_timeout (kombu.transport.redis.Channel.QoS attribute), 137
- visibility\_timeout (kombu.transport.redis.Transport.Channel attribute), 136
- visibility\_timeout (kombu.transport.redis.Transport.Channel.QoS attribute), 135
- visibility\_timeout (kombu.transport.SLMQ.Channel attribute), 147
- visibility\_timeout (kombu.transport.SLMQ.Transport.Channel attribute), 146
- visible (kombu.transport.django.models.Message attribute), 144

## W

- wait() (kombu.compat.Consumer method), 59
- wait() (kombu.transport.pyamqp.Connection method), 99
- wait() (kombu.transport.pyamqp.Connection.Channel method), 98
- warn() (kombu.log.LogMixin method), 64
- when\_bound() (kombu.abstract.MaybeChannelBound method), 75
- when\_bound() (kombu.Queue method), 44
- wildcards (kombu.transport.virtual.exchange.TopicExchange attribute), 156
- with\_metaclass() (in module kombu.five), 164
- WRITE (kombu.async.Hub attribute), 76
- WRITE (kombu.async.hub.Hub attribute), 77