py-amqp Documentation

Release 2.3.0

Ask Solem contributors

Apr 02, 2019

Contents

1 About **3**

2 Differences from amqplib **5**

3 Further **7**

4 Contents **9**

4.1 API Reference . . . 9
4.2 Changes . . . 61
4.3 2.3.0 . . . 61
4.4 2.2.2 . . . 62
4.5 2.2.1 . . . 62
4.6 2.2.0 . . . 63
4.7 2.1.4 . . . 63
4.8 2.1.3 . . . 64
4.9 2.1.2 . . . 64
4.10 2.1.1 . . . 64
4.11 2.1.0 . . . 65
4.12 2.0.3 . . . 65
4.13 2.0.2 . . . 66
4.14 2.0.1 . . . 66
4.15 2.0.0 . . . 66
4.16 1.4.9 . . . 67
4.17 1.4.8 . . . 67
4.18 1.4.7 . . . 67
4.19 1.4.6 . . . 68
4.20 1.4.5 . . . 68
4.21 1.4.4 . . . 68
4.22 1.4.3 . . . 68
4.23 1.4.2 . . . 69
4.24 1.4.1 . . . 69
4.25 1.4.0 . . . 69
4.26 1.3.3 . . . 69
4.27 1.3.2 . . . 70
4.28 1.3.1 . . . 70
4.29 1.3.0 . . . 70
4.30 1.2.1 . . . 71

4.31 1.2.0 . . . 71
4.32 1.1.0 . . . 72
4.33 1.0.13 . . . 72
4.34 1.0.12 . . . 72
4.35 1.0.11 . . . 72
4.36 1.0.10 . . . 72
4.37 1.0.9 . . . 73
4.38 1.0.8 . . . 73
4.39 1.0.7 . . . 73
4.40 1.0.6 . . . 73
4.41 1.0.5 . . . 73
4.42 1.0.4 . . . 74
4.43 1.0.3 . . . 74
4.44 1.0.2 . . . 74
4.45 1.0.1 . . . 74
4.46 1.0.0 . . . 74
4.47 Version 0.9.4 . . . 75
4.48 Version 0.9.3 . . . 75
4.49 Version 0.9.2 . . . 75
4.50 Version 0.9.1 . . . 75

5 Indices and tables **77**

Python Module Index **79**

Version 2.3.0

Web https://amqp.readthedocs.io/

Download https://pypi.org/project/amqp/

Source http://github.com/celery/py-amqp/

Keywords amqp, rabbitmq

CHAPTER 1

About

This is a fork of amqplib which was originally written by Barry Pederson. It is maintained by the Celery project, and used by kombu as a pure python alternative when librabbitmq is not available.

This library should be API compatible with librabbitmq.

CHAPTER 2

Differences from amqplib

- Supports draining events from multiple channels (`Connection.drain_events`)
- Support for timeouts
- Channels are restored after channel error, instead of having to close the connection.
- Support for heartbeats
  - `Connection.heartbeat_tick(rate=2)` must called at regular intervals (half of the heartbeat value if rate is 2).
  - Or some other scheme by using `Connection.send_heartbeat`.
- **Supports RabbitMQ extensions:**
  - **Consumer Cancel Notifications**
    - by default a cancel results in `ChannelError` being raised
    - but not if a `on_cancel` callback is passed to `basic_consume`.
  - **Publisher confirms**
    - `Channel.confirm_select()` enables publisher confirms.
    - `Channel.events['basic_ack'].append(my_callback)` adds a callback to be called when a message is confirmed. This callback is then called with the signature `(delivery_tag, multiple)`.
  - **Exchange-to-exchange bindings: `exchange_bind` / `exchange_unbind`.**
    - `Channel.confirm_select()` enables publisher confirms.
    - `Channel.events['basic_ack'].append(my_callback)` adds a callback to be called when a message is confirmed. This callback is then called with the signature `(delivery_tag, multiple)`.
- Support for `basic_return`
- **Uses AMQP 0-9-1 instead of 0-8.**

  - `Channel.access_request` and `ticket` arguments to methods **removed**.
  - Supports the `arguments` argument to `basic_consume`.
  - `internal` argument to `exchange_declare` removed.
  - `auto_delete` argument to `exchange_declare` deprecated
  - `insist` argument to `Connection` removed.
  - `Channel.alerts` has been removed.
  - Support for `Channel.basic_recover_async`.
  - `Channel.basic_recover` deprecated.
- **Exceptions renamed to have idiomatic names:**
  - `AMQPException` -> `AMQPError`
  - `AMQPConnectionException` -> ConnectionError‘‘
  - `AMQPChannelException` -> ChannelError‘‘
  - `Connection.known_hosts` removed.
  - `Connection` no longer supports redirects.
  - `exchange` argument to `queue_bind` can now be empty to use the “default exchange”.
- Adds `Connection.is_alive` that tries to detect whether the connection can still be used.
- Adds `Connection.connection_errors` and `.channel_errors`, a list of recoverable errors.
- Exposes the underlying socket as `Connection.sock`.
- Adds `Channel.no_ack_consumers` to keep track of consumer tags that set the no_ack flag.
- Slightly better at error recovery

CHAPTER 3

Further

- Differences between AMQP 0.8 and 0.9.1

  http://www.rabbitmq.com/amqp-0-8-to-0-9-1.html

- AMQP 0.9.1 Quick Reference

  http://www.rabbitmq.com/amqp-0-9-1-quickref.html

- RabbitMQ Extensions

  http://www.rabbitmq.com/extensions.html

- For more information about AMQP, visit

  http://www.amqp.org

- For other Python client libraries see:

  http://www.rabbitmq.com/devtools.html#python-dev

CHAPTER 4

Contents

4.1 API Reference

Release 2.3

Date Apr 02, 2019

4.1.1 amqp.connection

AMQP Connections.

class amqp.connection.**Connection**(*host='localhost:5672'*, *userid='guest'*, *password='guest'*, *login_method=None*, *login_response=None*, *authentication=()*, *virtual_host='/'*, *locale='en_US'*, *client_properties=None*, *ssl=False*, *connect_timeout=None*, *channel_max=None*, *frame_max=None*, *heartbeat=0*, *on_open=None*, *on_blocked=None*, *on_unblocked=None*, *confirm_publish=False*, *on_tune_ok=None*, *read_timeout=None*, *write_timeout=None*, *socket_settings=None*, *frame_handler=<function frame_handler>*, *frame_writer=<function frame_writer>*, ***kwargs*)

AMQP Connection.

The connection class provides methods for a client to establish a network connection to a server, and for both peers to operate the connection thereafter.

GRAMMAR:

```
connection          = open-connection *use-connection close-connection
open-connection     = C:protocol-header
                      S:START C:START-OK
                      *challenge
```

(continues on next page)

(continued from previous page)

```
                      S:TUNE C:TUNE-OK
                      C:OPEN S:OPEN-OK
challenge           = S:SECURE C:SECURE-OK
use-connection      = *channel
close-connection    = C:CLOSE S:CLOSE-OK
                    / S:CLOSE C:CLOSE-OK
```

Create a connection to the specified host, which should be a 'host[:port]', such as 'localhost', or '1.2.3.4:5672' (defaults to 'localhost', if a port is not specified then 5672 is used)

Authentication can be controlled by passing one or more *amqp.sasl.SASL* instances as the *authentication* parameter, or setting the *login_method* string to one of the supported methods: 'GSSAPI', 'EXTERNAL', 'AMQPLAIN', or 'PLAIN'. Otherwise authentication will be performed using any supported method preferred by the server. Userid and passwords apply to AMQPLAIN and PLAIN authentication, whereas on GSSAPI only userid will be used as the client name. For EXTERNAL authentication both userid and password are ignored.

The 'ssl' parameter may be simply True/False, or for Python >= 2.6 a dictionary of options to pass to ssl.wrap_socket() such as requiring certain certificates.

The "socket_settings" parameter is a dictionary defining tcp settings which will be applied as socket options.

class Channel(*connection*, *channel_id=None*, *auto_decode=True*, *on_open=None*)

AMQP Channel.

The channel class provides methods for a client to establish a virtual connection - a channel - to a server and for both peers to operate the virtual connection thereafter.

GRAMMAR:

```
channel             = open-channel *use-channel close-channel
open-channel        = C:OPEN S:OPEN-OK
use-channel         = C:FLOW S:FLOW-OK
                    / S:FLOW C:FLOW-OK
                    / functional-class
close-channel       = C:CLOSE S:CLOSE-OK
                    / S:CLOSE C:CLOSE-OK
```

Create a channel bound to a connection and using the specified numeric channel_id, and open on the server.

The 'auto_decode' parameter (defaults to True), indicates whether the library should attempt to decode the body of Messages to a Unicode string if there's a 'content_encoding' property for the message. If there's no 'content_encoding' property, or the decode raises an Exception, the message body is left as plain bytes.

basic_ack(*delivery_tag*, *multiple=False*, *argsig='Lb'*)

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 mesages.

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.

basic_cancel(*consumer_tag*, *nowait=False*, *argsig='sb'*)
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 abitrary 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.

basic_consume(*queue=''*, *consumer_tag=''*, *no_local=False*, *no_ack=False*, *exclusive=False*, *nowait=False*, *callback=None*, *arguments=None*, *on_cancel=None*, *argsig='BssbbbbF'*)
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 acknowledgment 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.

basic_get(*queue=''*, *no_ack=False*, *argsig='Bsb'*)
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 acknowledgment 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.

basic_publish(*msg*, *exchange=''*, *routing_key=''*, *mandatory=False*, *immediate=False*, *timeout=None*, *argsig='Bssbb'*)

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.

basic_publish_confirm(**args*, ***kwargs*)

basic_qos(*prefetch_size*, *prefetch_count*, *a_global*, *argsig='lBb'*)

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.

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.

basic_recover_async(*requeue=False*)

basic_reject(*delivery_tag*, *requeue*, *argsig='Lb'*)
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.

close(*reply_code=0*, *reply_text=''*, *method_sig=(0, 0)*, *argsig='BsBB'*)

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.

collect()
Tear down this object.

Best called after we've agreed to close with the server.

confirm_select(*nowait=False*)
Enable publisher confirms for this channel.

Note: This is 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.

exchange_bind(*destination*, *source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)
Bind 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.

exchange_declare(*exchange*, *type*, *passive=False*, *durable=False*, *auto_delete=True*, *nowait=False*, *arguments=None*, *argsig='BssbbbbbF'*)

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.

exchange_delete(*exchange*, *if_unused=False*, *nowait=False*, *argsig='Bsbb'*)
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.

exchange_unbind(*destination*, *source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)
Unbind 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.

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 oveflowing 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 exchangeing 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.

open()

Open a channel for use.

This method opens a virtual connection (a channel).

RULE:

This method MUST NOT be called when the channel is already open.

PARAMETERS: out_of_band: shortstr (DEPRECATED)

out-of-band settings

Configures out-of-band transfers on this channel. The syntax and meaning of this field will be formally defined at a later date.

queue_bind(*queue*, *exchange=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)

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.

queue_declare(*queue=''*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=True*, *nowait=False*, *arguments=None*, *argsig='BsbbbbbF'*)

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

queue_delete(*queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*, *argsig='Bsbbb'*)

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 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.

queue_purge(*queue=''*, *nowait=False*, *argsig='Bsb'*)

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

queue_unbind(*queue*, *exchange*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssF'*)

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.

then(*on_success*, *on_error=None*)

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.

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.

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.

Transport(*host*, *connect_timeout*, *ssl=False*, *read_timeout=None*, *write_timeout=None*, *socket_settings=None*, ***kwargs*)

blocking_read(*timeout=None*)

bytes_recv = 0
Number of successful reads from socket.

bytes_sent = 0
Number of successful writes to socket.

channel(*channel_id=None*, *callback=None*)
Create new channel.

Fetch a Channel object identified by the numeric channel_id, or create that object if it doesn't already exist.

channel_errors = (<class 'amqp.exceptions.ChannelError'>,)

client_heartbeat = None
Original heartbeat interval value proposed by client.

close(*reply_code=0*, *reply_text=''*, *method_sig=(0, 0)*, *argsig='BsBB'*)
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.

collect()

connect(*callback=None*)

connected

connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'OSError'>, <cl

drain_events(*timeout=None*)

frame_writer

heartbeat = None
Final heartbeat interval value (in float seconds) after negotiation

heartbeat_tick(*rate=2*)
Send heartbeat packets if necessary.

Raises:

~amqp.exceptions.ConnectionForvced: if none have been received recently.

Note: This should be called frequently, on the order of once per second.

Keyword Arguments: rate (int): Previously used, but ignored now.

is_alive()

last_heartbeat_received = 0
Time of last heartbeat received (in monotonic time, if available).

last_heartbeat_sent = 0
Time of last heartbeat sent (in monotonic time, if available).

library_properties = {'product': 'py-amqp', 'product_version': '2.3.0'}
These are sent to the server to announce what features we support, type of client etc.

negotiate_capabilities = {'authentication_failure_close': True, 'connection.blocked':
Mapping of protocol extensions to enable. The server will report these in server_properties[capabilities], and if a key in this map is present the client will tell the server to either enable or disable the capability depending on the value set in this map. For example with:

negotiate_capabilities = { 'consumer_cancel_notify': True,

}

The client will enable this capability if the server reports support for it, but if the value is False the client will disable the capability.

on_inbound_frame

on_inbound_method(*channel_id*, *method_sig*, *payload*, *content*)

prev_recv = None
Number of bytes received from socket at the last heartbeat check.

prev_sent = None
Number of bytes sent to socket at the last heartbeat check.

recoverable_channel_errors = (<class 'amqp.exceptions.RecoverableChannelError'>,)

recoverable_connection_errors = (<class 'amqp.exceptions.RecoverableConnectionError'>,

send_heartbeat()

server_capabilities

server_heartbeat = None
Original heartbeat interval proposed by server.

sock

then(*on_success*, *on_error=None*)

transport

4.1.2 amqp.channel

AMQP Channels.

class amqp.channel.**Channel**(*connection*, *channel_id=None*, *auto_decode=True*, *on_open=None*)
AMQP Channel.

The channel class provides methods for a client to establish a virtual connection - a channel - to a server and for both peers to operate the virtual connection thereafter.

GRAMMAR:

```
channel             = open-channel *use-channel close-channel
open-channel        = C:OPEN S:OPEN-OK
use-channel         = C:FLOW S:FLOW-OK
                    / S:FLOW C:FLOW-OK
                    / functional-class
close-channel       = C:CLOSE S:CLOSE-OK
                    / S:CLOSE C:CLOSE-OK
```

Create a channel bound to a connection and using the specified numeric channel_id, and open on the server.

The 'auto_decode' parameter (defaults to True), indicates whether the library should attempt to decode the body of Messages to a Unicode string if there's a 'content_encoding' property for the message. If there's no 'content_encoding' property, or the decode raises an Exception, the message body is left as plain bytes.

basic_ack(*delivery_tag*, *multiple=False*, *argsig='Lb'*)

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 mesages.

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.

basic_cancel(*consumer_tag*, *nowait=False*, *argsig='sb'*)

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 abitrary 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.

basic_consume(*queue=''*, *consumer_tag=''*, *no_local=False*, *no_ack=False*, *exclusive=False*, *nowait=False*, *callback=None*, *arguments=None*, *on_cancel=None*, *argsig='BssbbbbF'*)

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 acknowledgment 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.

basic_get(*queue=''*, *no_ack=False*, *argsig='Bsb'*)

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 acknowledgment 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.

basic_publish(*msg*, *exchange=''*, *routing_key=''*, *mandatory=False*, *immediate=False*, *timeout=None*, *argsig='Bssbb'*)

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.

basic_publish_confirm(**args*, ***kwargs*)

basic_qos(*prefetch_size*, *prefetch_count*, *a_global*, *argsig='lBb'*)

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.

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.

basic_recover_async(*requeue=False*)

basic_reject(*delivery_tag*, *requeue*, *argsig='Lb'*)

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.

close(*reply_code=0*, *reply_text=''*, *method_sig=(0, 0)*, *argsig='BsBB'*)

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.

collect()
Tear down this object.

Best called after we've agreed to close with the server.

confirm_select(*nowait=False*)
Enable publisher confirms for this channel.

Note: This is 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.

exchange_bind(*destination*, *source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)
Bind 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.

exchange_declare(*exchange*, *type*, *passive=False*, *durable=False*, *auto_delete=True*, *nowait=False*, *arguments=None*, *argsig='BssbbbbbF'*)

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.

exchange_delete(*exchange*, *if_unused=False*, *nowait=False*, *argsig='Bsbb'*)

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.

exchange_unbind(*destination*, *source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)

Unbind 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.

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 oveflowing 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 exchangeing 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.

open()

Open a channel for use.

This method opens a virtual connection (a channel).

RULE:

This method MUST NOT be called when the channel is already open.

PARAMETERS: out_of_band: shortstr (DEPRECATED)

out-of-band settings

Configures out-of-band transfers on this channel. The syntax and meaning of this field will be formally defined at a later date.

queue_bind(*queue*, *exchange=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)

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.

queue_declare(*queue=''*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=True*, *nowait=False*, *arguments=None*, *argsig='BsbbbbbF'*)

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

queue_delete(*queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*, *argsig='Bsbbb'*)

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 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.

queue_purge(*queue=''*, *nowait=False*, *argsig='Bsb'*)

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

queue_unbind(*queue*, *exchange*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssF'*)

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.

then(*on_success*, *on_error=None*)

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.

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.

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.

4.1.3 amqp.basic_message

AMQP Messages.

class amqp.basic_message.**Message**(*body=''*, *children=None*, *channel=None*, ***properties*)
A Message for use with the Channel.basic_* methods.

Expected arg types

body: string children: (not supported)

Keyword properties may include:

content_type: shortstr MIME content type

content_encoding: shortstr MIME content encoding

application_headers: table Message header field table, a dict with string keys, and string | int | Decimal | datetime | dict values.

delivery_mode: octet Non-persistent (1) or persistent (2)

priority: octet The message priority, 0 to 9

correlation_id: shortstr The application correlation identifier

reply_to: shortstr The destination to reply to

expiration: shortstr Message expiration specification

message_id: shortstr The application message identifier

timestamp: datetime.datetime The message timestamp

type: shortstr The message type name

user_id: shortstr The creating user id

app_id: shortstr The creating application id

cluster_id: shortstr Intra-cluster routing identifier

Unicode bodies are encoded according to the 'content_encoding' argument. If that's None, it's set to 'UTF-8' automatically.

Example:

```
msg = Message('hello world',
                content_type='text/plain',
                application_headers={'foo': 7})
```

CLASS_ID = 60

PROPERTIES = [('content_type', 's'), ('content_encoding', 's'), ('application_headers'
Instances of this class have these attributes, which are passed back and forth as message properties between client and server

delivery_info = None
set by basic_consume/basic_get

delivery_tag

headers

4.1.4 amqp.exceptions

Exceptions used by amqp.

exception amqp.exceptions.**AMQPError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

Base class for all AMQP exceptions.

code = 0

method

exception amqp.exceptions.**ConnectionError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Connection Error.

exception amqp.exceptions.**ChannelError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Channel Error.

exception amqp.exceptions.**RecoverableConnectionError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

Exception class for recoverable connection errors.

exception amqp.exceptions.**IrrecoverableConnectionError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

Exception class for irrecoverable connection errors.

exception amqp.exceptions.**RecoverableChannelError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

Exception class for recoverable channel errors.

exception amqp.exceptions.**IrrecoverableChannelError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

Exception class for irrecoverable channel errors.

exception amqp.exceptions.**ConsumerCancelled**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Consumer Cancelled Predicate.

exception amqp.exceptions.**ContentTooLarge**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Content Too Large Error.

code = 311

exception amqp.exceptions.**NoConsumers**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP No Consumers Error.

code = 313

exception amqp.exceptions.**ConnectionForced**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Connection Forced Error.

code = 320

exception amqp.exceptions.**InvalidPath**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Invalid Path Error.

code = 402

exception amqp.exceptions.**AccessRefused**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Access Refused Error.

code = 403

exception amqp.exceptions.**NotFound**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Not Found Error.

code = 404

exception amqp.exceptions.**ResourceLocked**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Resource Locked Error.

code = 405

exception amqp.exceptions.**PreconditionFailed**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Precondition Failed Error.

code = 406

exception amqp.exceptions.**FrameError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Frame Error.

code = 501

exception amqp.exceptions.**FrameSyntaxError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Frame Syntax Error.

code = 502

exception amqp.exceptions.**InvalidCommand**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Invalid Command Error.

code = 503

exception amqp.exceptions.**ChannelNotOpen**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Channel Not Open Error.

code = 504

exception amqp.exceptions.**UnexpectedFrame**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)

AMQP Unexpected Frame.

code = 505

exception amqp.exceptions.**ResourceError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)
AMQP Resource Error.

code = 506

exception amqp.exceptions.**NotAllowed**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)
AMQP Not Allowed Error.

code = 530

exception amqp.exceptions.**AMQPNotImplementedError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)
AMQP Not Implemented Error.

code = 540

exception amqp.exceptions.**InternalError**(*reply_text=None*, *method_sig=None*, *method_name=None*, *reply_code=None*)
AMQP Internal Error.

code = 541

exception amqp.exceptions.**AMQPDeprecationWarning**
Warning for deprecated things.

4.1.5 amqp.abstract_channel

Code common to Connection and Channel objects.

class amqp.abstract_channel.**AbstractChannel**(*connection*, *channel_id*)
Superclass for Connection and Channel.

The connection is treated as channel 0, then comes user-created channel objects.

The subclasses must have a _METHOD_MAP class property, mapping between AMQP method signatures and Python methods.

close()
Close this Channel or Connection.

dispatch_method(*method_sig*, *payload*, *content*)

send_method(*sig*, *format=None*, *args=None*, *content=None*, *wait=None*, *callback=None*, *returns_tuple=False*)

wait(*method*, *callback=None*, *timeout=None*, *returns_tuple=False*)

4.1.6 amqp.transport

Transport implementation.

class amqp.transport.**SSLTransport**(*host*, *connect_timeout=None*, *ssl=None*, ***kwargs*)
Transport that works over SSL.

class amqp.transport.**TCPTransport**(*host*, *connect_timeout=None*, *read_timeout=None*, *write_timeout=None*, *socket_settings=None*, *raise_on_initial_eintr=True*, ***kwargs*)
Transport that deals directly with TCP socket.

amqp.transport.**Transport**(*host*, *connect_timeout=None*, *ssl=False*, ***kwargs*)
Create transport.

Given a few parameters from the Connection constructor, select and create a subclass of _AbstractTransport.

amqp.transport.**to_host_port**(*host*, *default=5672*)
Convert hostname:port string to host, port tuple.

4.1.7 amqp.method_framing

Convert between frames and higher-level AMQP methods.

amqp.method_framing.**frame_handler**(*connection*, *callback*, *unpack_from=<built-in function unpack_from>*, *content_methods=frozenset({(60, 50), (60, 71), (60, 60)})*)
Create closure that reads frames.

amqp.method_framing.**frame_writer**(*connection*, *transport*, *pack=<built-in function pack>*, *pack_into=<built-in function pack_into>*, *range=<class 'range'>*, *len=<built-in function len>*, *bytes=<class 'bytes'>*, *str_to_bytes=<function str_to_bytes>*)
Create closure that writes frames.

4.1.8 amqp.platform

Platform compatibility.

amqp.platform.**pack**(*format*, *v1*, *v2*, ...) → bytes
Return a bytes object containing the values v1, v2, … packed according to the format string. See help(struct) for more on format strings.

amqp.platform.**pack_into**(*format*, *buffer*, *offset*, *v1*, *v2*, ...)
Pack the values v1, v2, … according to the format string and write the packed bytes into the writable buffer buf starting at offset. Note that the offset is a required argument. See help(struct) for more on format strings.

amqp.platform.**unpack**()
Return a tuple containing values unpacked according to the format string.

The buffer's size in bytes must be calcsize(format).

See help(struct) for more on format strings.

amqp.platform.**unpack_from**()
Return a tuple containing values unpacked according to the format string.

The buffer's size, minus offset, must be at least calcsize(format).

See help(struct) for more on format strings.

4.1.9 amqp.protocol

Protocol data.

class amqp.protocol.**basic_return_t**(*reply_code*, *reply_text*, *exchange*, *routing_key*, *message*)

exchange
Alias for field number 2

message
Alias for field number 4

reply_code
Alias for field number 0

reply_text
Alias for field number 1

routing_key
Alias for field number 3

class amqp.protocol.**queue_declare_ok_t**(*queue*, *message_count*, *consumer_count*)

consumer_count
Alias for field number 2

message_count
Alias for field number 1

queue
Alias for field number 0

4.1.10 amqp.spec

SASL mechanisms for AMQP authentication.

class amqp.sasl.**AMQPLAIN**(*username*, *password*)
AMQPLAIN SASL authentication mechanism.

This is a non-standard mechanism used by AMQP servers.

mechanism = b'AMQPLAIN'

start(*connection*)
Return the first response to a SASL challenge as a bytes object.

class amqp.sasl.**EXTERNAL**
EXTERNAL SASL mechanism.

Enables external authentication, i.e. not handled through this protocol. Only passes 'EXTERNAL' as authentication mechanism, but no further authentication data.

mechanism = b'EXTERNAL'

start(*connection*)
Return the first response to a SASL challenge as a bytes object.

amqp.sasl.**GSSAPI**
alias of amqp.sasl._get_gssapi_mechanism.<locals>.FakeGSSAPI

class amqp.sasl.**PLAIN**(*username*, *password*)
PLAIN SASL authentication mechanism.

See https://tools.ietf.org/html/rfc4616 for details

mechanism = b'PLAIN'

start(*connection*)
Return the first response to a SASL challenge as a bytes object.

class amqp.sasl.**RAW**(*mechanism*, *response*)
A generic custom SASL mechanism.

This mechanism takes a mechanism name and response to send to the server, so can be used for simple custom authentication schemes.

mechanism = None

start(*connection*)
Return the first response to a SASL challenge as a bytes object.

class amqp.sasl.**SASL**
The base class for all amqp SASL authentication mechanisms.

You should sub-class this if you're implementing your own authentication.

mechanism
Return a bytes containing the SASL mechanism name.

start(*connection*)
Return the first response to a SASL challenge as a bytes object.

4.1.11 amqp.serialization

Convert between bytestreams and higher-level AMQP types.

2007-11-05 Barry Pederson <bp@barryp.org>

class amqp.serialization.**GenericContent**(*frame_method=None*, *frame_args=None*, ***props*)
Abstract base class for AMQP content.

Subclasses should override the PROPERTIES attribute.

CLASS_ID = None

PROPERTIES = [('dummy', 's')]

inbound_body(*buf*)

inbound_header(*buf*, *offset=0*)

amqp.serialization.**decode_properties_basic**(*buf*, *offset=0*, *unpack_from=<built-in function unpack_from>*, *pstr_t=<function bytes_to_str>*)
Decode basic properties.

amqp.serialization.**dumps**(*format*, *values*)
Serialize AMQP arguments.

Notes: bit = b octet = o short = B long = l long long = L shortstr = s longstr = S table = F array = A

amqp.serialization.**loads**(*format*, *buf*, *offset=0*, *ord=<built-in function ord>*, *unpack_from=<built-in function unpack_from>*, *_read_item=<function _read_item>*, *pstr_t=<function bytes_to_str>*)
Deserialize amqp format.

bit = b octet = o short = B long = l long long = L float = f shortstr = s longstr = S table = F array = A timestamp = T

4.1.12 amqp.spec

AMQP Spec.

class amqp.spec.**Basic**
AMQ Basic class.

Ack = (60, 80)

CLASS_ID = 60

Cancel = (60, 30)

CancelOk = (60, 31)

Consume = (60, 20)

ConsumeOk = (60, 21)

Deliver = (60, 60)

Get = (60, 70)

GetEmpty = (60, 72)

GetOk = (60, 71)

Nack = (60, 120)

Publish = (60, 40)

Qos = (60, 10)

QosOk = (60, 11)

Recover = (60, 110)

RecoverAsync = (60, 100)

RecoverOk = (60, 111)

Reject = (60, 90)

Return = (60, 50)

class amqp.spec.**Channel**
AMQ Channel class.

CLASS_ID = 20

Close = (20, 40)

CloseOk = (20, 41)

Flow = (20, 20)

FlowOk = (20, 21)

Open = (20, 10)

OpenOk = (20, 11)

class amqp.spec.**Confirm**
AMQ Confirm class.

CLASS_ID = 85

Select = (85, 10)

SelectOk = (85, 11)

class amqp.spec.**Connection**
AMQ Connection class.

Blocked = (10, 60)

CLASS_ID = 10

Close = (10, 50)

CloseOk = (10, 51)

Open = (10, 40)

OpenOk = (10, 41)

Secure = (10, 20)

SecureOk = (10, 21)

Start = (10, 10)

StartOk = (10, 11)

Tune = (10, 30)

TuneOk = (10, 31)

Unblocked = (10, 61)

class amqp.spec.**Exchange**
AMQ Exchange class.

Bind = (40, 30)

BindOk = (40, 31)

CLASS_ID = 40

Declare = (40, 10)

DeclareOk = (40, 11)

Delete = (40, 20)

DeleteOk = (40, 21)

Unbind = (40, 40)

UnbindOk = (40, 51)

class amqp.spec.**Queue**
AMQ Queue class.

Bind = (50, 20)

BindOk = (50, 21)

CLASS_ID = 50

Declare = (50, 10)

DeclareOk = (50, 11)

Delete = (50, 40)

DeleteOk = (50, 41)

Purge = (50, 30)

PurgeOk = (50, 31)

Unbind = (50, 50)

UnbindOk = (50, 51)

class amqp.spec.**Tx**
AMQ Tx class.

CLASS_ID = 90

Commit = (90, 20)

CommitOk = (90, 21)

Rollback = (90, 30)

RollbackOk = (90, 31)

Select = (90, 10)

SelectOk = (90, 11)

amqp.spec.**method**(*method_sig*, *args=None*, *content=False*)
Create amqp method specification tuple.

class amqp.spec.**method_t**(*method_sig*, *args*, *content*)

args
Alias for field number 1

content
Alias for field number 2

method_sig
Alias for field number 0

4.1.13 amqp.utils

Compatibility utilities.

class amqp.utils.**NullHandler**(*level=0*)
A logging handler that does nothing.

emit(*record*)
Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

amqp.utils.**bytes_to_str**(*s*)
Convert bytes to str.

amqp.utils.**coro**(*gen*)
Decorator to mark generator as a co-routine.

amqp.utils.**get_errno**(*exc*)
Get exception errno (if set).

Notes: socket.error and IOError first got the .errno attribute in Py2.7.

amqp.utils.**get_logger**(*logger*)
Get logger by name.

amqp.utils.**set_cloexec**(*fd*, *cloexec*)
Set flag to close fd after exec.

amqp.utils.**str_to_bytes**(*s*)
Convert str to bytes.

4.1.14 amqp.five

Python 2/3 compatibility.

Compatibility implementations of features only available in newer Python versions.

class amqp.five.**Counter**(***kwds*)
Dict subclass for counting hashable items. Sometimes called a bag or multiset. Elements are stored as dictionary keys and their counts are stored as dictionary values.

```
>>> c = Counter('abcdeabcdabcaba')  # count elements from a string
```

```
>>> c.most_common(3)                # three most common elements
[('a', 5), ('b', 4), ('c', 3)]
>>> sorted(c)                       # list all unique elements
['a', 'b', 'c', 'd', 'e']
>>> ''.join(sorted(c.elements()))   # list elements with repetitions
'aaaaabbbbcccdde'
>>> sum(c.values())                 # total of all counts
15
```

```
>>> c['a']                          # count of letter 'a'
5
>>> for elem in 'shazam':           # update counts from an iterable
...     c[elem] += 1                # by adding 1 to each element's count
>>> c['a']                          # now there are seven 'a'
7
>>> del c['b']                      # remove all 'b'
>>> c['b']                          # now there are zero 'b'
0
```

```
>>> d = Counter('simsalabim')       # make another counter
>>> c.update(d)                     # add in the second counter
>>> c['a']                          # now there are nine 'a'
9
```

```
>>> c.clear()                       # empty the counter
>>> c
Counter()
```

Note: If a count is set to zero or reduced to zero, it will remain in the counter until the entry is deleted or the counter is cleared:

```
>>> c = Counter('aaabbc')
>>> c['b'] -= 2                     # reduce the count of 'b' by two
>>> c.most_common()                 # 'b' is still in, but its count is zero
[('a', 3), ('c', 1), ('b', 0)]
```

copy()
Return a shallow copy.

elements()
Iterator over elements repeating each as many times as its count.

```
>>> c = Counter('ABCABC')
>>> sorted(c.elements())
['A', 'A', 'B', 'B', 'C', 'C']
```

Knuth's example for prime factors of 1836: 2**2 * 3**3 * 17**1 >>> prime_factors = Counter({2: 2, 3: 3, 17: 1}) >>> product = 1 >>> for factor in prime_factors.elements(): # loop over factors ... product *= factor # and multiply them >>> product 1836

Note, if an element's count has been set to zero or is a negative number, elements() will ignore it.

classmethod fromkeys(*iterable*, *v=None*)
Create a new dictionary with keys from iterable and values set to value.

most_common(*n=None*)
List the n most common elements and their counts from the most common to the least. If n is None, then list all element counts.

```
>>> Counter('abcdeabcdabcaba').most_common(3)
[('a', 5), ('b', 4), ('c', 3)]
```

subtract(***kwds*)
Like dict.update() but subtracts counts instead of replacing them. Counts can be reduced below zero. Both the inputs and outputs are allowed to contain zero and negative counts.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.subtract('witch')             # subtract elements from another iterable
>>> c.subtract(Counter('watch'))    # subtract elements from another counter
>>> c['h']                          # 2 in which, minus 1 in witch, minus 1␣
↪in watch
0
>>> c['w']                          # 1 in which, minus 1 in witch, minus 1␣
↪in watch
-1
```

update(***kwds*)
Like dict.update() but add counts instead of replacing them.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.update('witch')           # add elements from another iterable
>>> d = Counter('watch')
>>> c.update(d)                 # add elements from another counter
>>> c['h']                      # four 'h' in which, witch, and watch
4
```

amqp.five.**reload**(*module*)
Reload the module and return it.

The module must have been successfully imported before.

class amqp.five.**UserList**(*initlist=None*)
A more or less complete user-defined wrapper around list objects.

append(*item*)
S.append(value) – append value to the end of the sequence

clear() → None – remove all items from S

copy()

count(*value*) → integer – return number of occurrences of value

extend(*other*)
S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*i*, *item*)
S.insert(index, value) – insert value before index

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*item*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

sort(**args*, ***kwds*)

class amqp.five.**UserDict**(***kwargs*)

copy()

classmethod fromkeys(*iterable*, *value=None*)

class amqp.five.**Callable**

class amqp.five.**Iterable**

class amqp.five.**Mapping**

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

class amqp.five.**Queue**(*maxsize=0*)
Create a queue object with a given maximum size.

If maxsize is <= 0, the queue size is infinite.

empty()
Return True if the queue is empty, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0 as a direct substitute, but be aware that either approach risks a race condition where a queue can grow before the result of empty() or qsize() can be used.

To create code that needs to wait for all queued tasks to be completed, the preferred technique is to use the join() method.

full()
Return True if the queue is full, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() >= n as a direct substitute, but be aware that either approach risks a race condition where a queue can shrink before the result of full() or qsize() can be used.

get(*block=True*, *timeout=None*)
Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until an item is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Empty exception if no item was available within that time. Otherwise ('block' is false), return an item if one is immediately available, else raise the Empty exception ('timeout' is ignored in that case).

get_nowait()
Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

join()
Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

put(*item*, *block=True*, *timeout=None*)
Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

put_nowait(*item*)
Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

qsize()
Return the approximate size of the queue (not reliable!).

task_done()
Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

exception amqp.five.**Empty**
Exception raised by Queue.get(block=0)/get_nowait().

exception amqp.five.**Full**
Exception raised by Queue.put(block=0)/put_nowait().

class amqp.five.**LifoQueue**(*maxsize=0*)
Variant of Queue that retrieves most recently added entries first.

class amqp.five.**array**(*typecode*[, *initializer*]) → array

Return a new array whose items are restricted by typecode, and initialized from the optional initializer value, which must be a list, string or iterable over elements of the appropriate type.

Arrays represent basic values and behave very much like lists, except the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:

Type code C Type Minimum size in bytes 'b' signed integer 1 'B' unsigned integer 1 'u' Unicode character 2 (see note) 'h' signed integer 2 'H' unsigned integer 2 'i' signed integer 2 'I' unsigned integer 2 'l' signed integer 4 'L' unsigned integer 4 'q' signed integer 8 (see note) 'Q' unsigned integer 8 (see note) 'f' floating point 4 'd' floating point 8

NOTE: The 'u' typecode corresponds to Python's unicode character. On narrow builds this is 2-bytes on wide builds this is 4-bytes.

NOTE: The 'q' and 'Q' type codes are only available if the platform C compiler used to build Python supports 'long long', or, on Windows, '__int64'.

Methods:

append() – append a new item to the end of the array buffer_info() – return information giving the current memory info byteswap() – byteswap all the items of the array count() – return number of occurrences of an object extend() – extend array by appending multiple elements from an iterable fromfile() – read items from a file object fromlist() – append items from the list frombytes() – append items from the string index() – return index of first occurrence of an object insert() – insert a new item into the array at a provided position pop() – remove and return item (default last) remove() – remove first occurrence of an object reverse() – reverse the order of the items in the array tofile() – write all items to a file object tolist() – return the array converted to an ordinary list tobytes() – return the array converted to a string

Attributes:

typecode – the typecode character used to create the array itemsize – the length in bytes of one array item

append()
Append new value v to the end of the array.

buffer_info()
Return a tuple (address, length) giving the current memory address and the length in items of the buffer used to hold array's contents.

The length should be multiplied by the itemsize attribute to calculate the buffer length in bytes.

byteswap()
Byteswap all items of the array.

If the items in the array are not 1, 2, 4, or 8 bytes in size, RuntimeError is raised.

count()
Return number of occurrences of v in the array.

extend()
Append items to the end of the array.

frombytes()
Appends items from the string, interpreting it as an array of machine values, as if it had been read from a file using the fromfile() method).

fromfile()
Read n objects from the file object f and append them to the end of the array.

fromlist()
Append items to array from list.

fromstring()
Appends items from the string, interpreting it as an array of machine values, as if it had been read from a file using the fromfile() method).

This method is deprecated. Use frombytes instead.

fromunicode()
Extends this array with data from the unicode string ustr.

The array must be a unicode type array; otherwise a ValueError is raised. Use array.frombytes(ustr.encode(...)) to append Unicode data to an array of some other type.

index()
Return index of first occurrence of v in the array.

insert()
Insert a new item v into the array before position i.

itemsize
the size, in bytes, of one array item

pop()
Return the i-th element and delete it from the array.

i defaults to -1.

remove()
Remove the first occurrence of v in the array.

reverse()
Reverse the order of the items in the array.

tobytes()
Convert the array to an array of machine values and return the bytes representation.

tofile()
Write all items (as machine values) to the file object f.

tolist()
Convert array to an ordinary list with the same items.

tostring()
Convert the array to an array of machine values and return the bytes representation.

This method is deprecated. Use tobytes instead.

tounicode()
Extends this array with data from the unicode string ustr.

Convert the array to a unicode string. The array must be a unicode type array; otherwise a ValueError is raised. Use array.tobytes().decode() to obtain a unicode string from an array of some other type.

typecode
the typecode character used to create the array

class amqp.five.**zip_longest**
zip_longest(iter1 [,iter2 [...]], [fillvalue=None]) –> zip_longest object

Return a zip_longest object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the longest iterable in the argument sequence is exhausted and then it raises StopIteration. When the shorter iterables are exhausted, the fillvalue is substituted in their place. The fillvalue defaults to None or can be specified by a keyword argument.

class amqp.five.**map**
map(func, *iterables) –> map object

Make an iterator that computes the function using arguments from each of the iterables. Stops when the shortest iterable is exhausted.

class amqp.five.**zip**
zip(iter1 [,iter2 [...]]) –> zip object

Return a zip object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the shortest iterable in the argument sequence is exhausted and then it raises StopIteration.

amqp.five.**string**
alias of builtins.str

amqp.five.**string_t**
alias of builtins.str

amqp.five.**bytes_t**
alias of builtins.bytes

amqp.five.**bytes_if_py2**(*s*)
Convert str to bytes if running under Python 2.

amqp.five.**long_t**
alias of builtins.int

amqp.five.**text_t**
alias of builtins.str

amqp.five.**module_name_t**
alias of builtins.str

class amqp.five.**range**(*stop*) → range object
range(start, stop[, step]) -> range object

Return an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ..., j-1. start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3. These are exactly the valid indices for a list of 4 elements. When step is given, it specifies the increment (or decrement).

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return index of value.
Raise ValueError if the value is not present.

start

step

stop

amqp.five.**items**(*d*)
Get dict items iterator.

amqp.five.**keys**(*d*)
Get dict keys iterator.

amqp.five.**values**(*d*)
Get dict values iterator.

amqp.five.**nextfun**(*it*)
Get iterator next method.

amqp.five.**reraise**(*tp*, *value*, *tb=None*)
Reraise exception.

class amqp.five.**WhateverIO**(*v=None*, **a*, ***kw*)
StringIO that takes bytes or str.

write(*data*)
Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

amqp.five.**with_metaclass**(*Type*, *skip_attrs=None*)
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).

class amqp.five.**StringIO**
Text I/O implementation using an in-memory buffer.

The initial_value argument sets the value of object. The newline argument is like the one of TextIOWrapper's constructor.

close()
Close the IO object.

Attempting any further operation after the object is closed will raise a ValueError.

This method has no effect if the file is already closed.

closed

getvalue()
Retrieve the entire contents of the object.

line_buffering

newlines
Line endings translated so far.

Only line endings translated during reading are considered.

Subclasses should override.

read()
Read at most size characters, returned as a string.

If the argument is negative or omitted, read until EOF is reached. Return an empty string at EOF.

readable()
Returns True if the IO object can be read.

readline()
Read until newline or EOF.

Returns an empty string if EOF is hit immediately.

seek()
Change stream position.

Seek to character offset pos relative to position indicated by whence: 0 Start of stream (the default). pos should be >= 0; 1 Current position - pos must be 0; 2 End of stream - pos must be 0.

Returns the new absolute position.

seekable()
Returns True if the IO object can be seeked.

tell()
Tell the current file position.

truncate()
Truncate size to pos.

The pos argument defaults to the current file position, as returned by tell(). The current file position is unchanged. Returns the new absolute position.

writable()
Returns True if the IO object can be written.

write()
Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

amqp.five.**getfullargspec**(*func*)
Get the names and default values of a callable object's parameters.

A tuple of seven things is returned: (args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations). 'args' is a list of the parameter names. 'varargs' and 'varkw' are the names of the * and ** parameters or None. 'defaults' is an n-tuple of the default values of the last n parameters. 'kwonlyargs' is a list of keyword-only parameter names. 'kwonlydefaults' is a dictionary mapping names from kwonlyargs to defaults. 'annotations' is a dictionary mapping parameter names to annotations.

Notable differences from inspect.signature():

- the "self" parameter is always reported, even for bound methods
- wrapper chains defined by __wrapped__ *not* unwrapped automatically

amqp.five.**format_d**(*i*)
Format number.

amqp.five.**monotonic**() → float
Monotonic clock, cannot go backward.

class amqp.five.**buffer_t**
Python 3 does not have a buffer type.

amqp.five.**python_2_unicode_compatible**(*cls*)
Class decorator to ensure class is compatible with Python 2.

4.2 Changes

py-amqp is fork of amqplib used by Kombu containing additional features and improvements. The previous amqplib changelog is here: http://code.google.com/p/py-amqplib/source/browse/CHANGES

4.3 2.3.0

release-date 2018-05-27 16:30 P.M UTC+3

release-by Omer Katz

- Cleanup TCP configurations across platforms and unified defaults.

  Fix contributed by **Dan Chowdhury**

- Fix for TypeError when setting socket options.

  Fix contributed by **Matthias Erll**

- Ensure that all call sites for decoding bytes to str allow surrogates, as the encoding mechanism now supports.

  Fix contributed by **Stephen Hatch**

- Don't send AAAA DNS request when domain resolved to IPv4 address.

  Fix contributed by **Ihar Hrachyshka & Omer Katz**

- Support for EXTERNAL authentication and specific login_method.

  Fix contributed by **Matthias Erll**

- If the old python-gssapi library is installed the gssapi module will be available. We now ensure that we only use the new gssapi library.

  Fix contributed by **Jacopo Notarstefano**

Code Cleanups & Test Coverage:

- @eric-eric-eric
- **Omer Katz**
- **Jon Dufresne**
- **Matthias Urlichs**

4.4 2.2.2

release-date 2017-09-14 09:00 A.M UTC+2

release-by Omer Katz

- Sending empty messages no longer hangs. Instead an empty message is sent correctly.(addresses #151)

  Fix contributed by **Christian Blades**

- Fixed compatibility issues in UTF-8 encoding behavior between Py2/Py3 (#164)

  Fix contributed by **Tyler James Harden**

4.5 2.2.1

release-date 2017-07-14 09:00 A.M UTC+2

release-by Omer Katz

- Fix implicit conversion from bytes to string on the connection object. (Issue #155)

  This issue has caused Celery to crash on connection to RabbitMQ.

  Fix contributed by **Omer Katz**

4.6 2.2.0

release-date 2017-07-12 10:00 A.M UTC+2

release-by Ask Solem

- Fix random delays in task execution.

  This is a bug that caused performance issues due to polling timeouts that occur when receiving incomplete AMQP frames. (Issues #3978 #3737 #3814)

  Fix contributed by **Robert Kopaczewski**

- Calling `conn.collect()` multiple times will no longer raise an `AttributeError` when no channels exist.

  Fix contributed by **Gord Chung**

- Fix compatibility code for Python 2.7.6.

  Fix contributed by **Jonathan Schuff**

- When running in Windows, py-amqp will no longer use the unsupported TCP option TCP_MAXSEG.

  Fix contributed by **Tony Breeds**

- Added support for setting the SNI hostname header.

  The SSL protocol version is now set to SSLv23

  Contributed by **Dhananjay Sathe**

- Authentication mechanisms were refactored to be more modular. GSSAPI authentication is now supported.

  Contributed by **Alexander Dutton**

- Do not reconnect on collect.

  Fix contributed by **Gord Chung**

4.7 2.1.4

release-date 2016-12-14 03:40 P.M PST

release-by Ask Solem

- Removes byte string comparison warnings when running under `python -b`.

  Fix contributed by **Jon Dufresne**.

- Linux version parsing broke when the version included a '+' character (Issue #119).

- Now sets default TCP settings for platforms that support them (e.g. Linux).

Constant	Value
`TCP_KEEPIDLE`	`60`
`TCP_KEEPINTVL`	`10`
`TCP_KEEPCNT`	`9`
`TCP_USER_TIMEOUT`	`1000` (1s)

  This will help detecting the socket being closed earlier, which is very important in failover and load balancing scenarios.

4.8 2.1.3

release-date 2016-12-07 06:00 P.M PST

release-by Ask Solem

- Fixes compatibility with Python 2.7.5 and below (Issue #107).

4.9 2.1.2

release-date 2016-12-07 02:00 P.M PST

- Linux: Now sets the `TCP_USER_TIMEOUT` flag if available for better failed connection detection.

  Contributed by **Jelte Fennema**.

  The timeout is set to the `connect_timeout` value by default, but can also be specified by using the `socket_settings` argument to `Connection`:

```
from amqp import Connection
from amqp.platform import TCP_USER_TIMEOUT

conn = Connection(socket_settings={
    TCP_USER_TIMEOUT: int(60 * 1000),  # six minutes in ms.
})
```

  When using Kombu this can be specified as part of the `transport_options`:

```
from amqp.platform import TCP_USER_TIMEOUT
from kombu import Connection

conn = Connection(transport_options={
    'socket_settings': {
        TCP_USER_TIMEOUT: int(60 * 1000),  # six minutes in ms.
    },
})
```

  Or when using Celery it can be specified using the `broker_transport_options` setting:

```
from amqp.platform import TCP_USER_TIMEOUT
from celery import Celery

app = Celery()
app.conf.update(
    broker_transport_options={
        TCP_USER_TIMEOUT: int(60 * 1000),  # six minutes in ms.
    }
)
```

- Python compatibility: Fixed compatibility when using the python `-b` flag.

  Fix contributed by Jon Dufresne.

4.10 2.1.1

release-date 2016-10-13 06:36 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on Vine 1.1.3.
- Frame writer: Account for overhead when calculating frame size.

  The client would crash if the message was within a certain size.
- Fixed struct unicode problems (#108)
  - Standardize pack invocations on bytestrings.
  - Leave some literals as strings to enable interpolation.
  - Fix flake8 fail.

  Fix contributed by **Brendan Smithyman**.

4.11 2.1.0

release-date 2016-09-07 04:23 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on Vine 1.1.2.
- Now licensed under the BSD license!

  Thanks to Barry Pederson for approving the license change, which unifies the license used across all projects in the Celery organization.
- Datetimes in method frame arguments are now handled properly.
- Fixed compatibility with Python <= 2.7.6
- Frame_writer is no longer a generator, which should solve a rare "generator already executing" error (Issue #103).

4.12 2.0.3

release-date 2016-07-11 08:00 P.M PDT

release-by Ask Solem

- SSLTransport: Fixed crash "no attribute sslopts" when `ssl=True` (Issue #100).
- Fixed incompatible argument spec for `Connection.Close` (Issue #45).

  This caused the RabbitMQ server to raise an exception (INTERNAL ERROR).
- Transport: No longer implements *__del__* to make sure gc can collect connections.

  It's the responsibility of the caller to close connections, this was simply a relic from the amqplib library.

4.13 2.0.2

release-date 2016-06-10 5:40 P.M PDT

release-by Ask Solem

- Python 3: Installation requirements ended up being a generator and crashed setup.py.

  Fix contributed by ChangBo Guo(gcb).

- Python <= 2.7.7: struct.pack arguments cannot be unicode

  Fix contributed by Alan Justino and Xin Li.

- Python 3.4: Fixed use of *bytes % int*.

  Fix contributed by Alan Justino.

- Connection/Transport: Fixed handling of default port.

  Fix contributed by Quentin Pradet.

4.14 2.0.1

release-date 2016-05-31 6:20 P.M PDT

release-by Ask Solem

- Adds backward compatibility layer for the 1.4 API.

  Using the connection without calling `.connect()` first will now work, but a warning is emitted and the behavior is deprecated and will be removed in version 2.2.

- Fixes kombu 3.0/celery 3.1 compatibility (Issue #88).

  Fix contributed by Bas ten Berge.

- Fixed compatibility with Python 2.7.3 (Issue #85)

  Fix contributed by Bas ten Berge.

- Fixed bug where calling drain_events() with a timeout of 0 would actually block until a frame is received.

- Documentation moved to http://amqp.readthedocs.io (Issue #89).

  See https://blog.readthedocs.com/securing-subdomains/ for the reasoning behind this change.

  Fix contributed by Adam Chainz.

4.15 2.0.0

release-date 2016-05-26 1:44 P.M PDT

release-by Ask Solem

- No longer supports Python 2.6

- You must now call Connection.connect() to establish the connection.

  The Connection constructor no longer has side effects, so you have to explicitly call connect first.

- Library rewritten to anticipate async changes.

- Connection now exposes underlying socket options.

  This change allows to set arbitrary TCP socket options during the creation of the transport.

  Those values can be set passing a dictionray where the key is the name of the parameter we want to set. The names of the keys are the ones reported above.

  Contributed by Andrea Rosa, Dallas Marlow and Rongze Zhu.

- Additional logging for heartbeats.

  Contributed by Davanum Srinivas, and Dmitry Mescheryakov.

- SSL: Fixes issue with remote connection hanging

  Fix contributed by Adrien Guinet.

- **SSL: `ssl` dict argument now supports the `check_hostname` key** (Issue #63).

  Contributed by Vic Kumar.

- Contributions by:

  Adrien Guinet Andrea Rosa Artyom Koval Corey Farwell Craig Jellick Dallas Marlow Davanum Srinivas Federico Ficarelli Jared Lewis Rémy Greinhofer Rongze Zhu Yury Selivanov Vic Kumar Vladimir Bolshakov @lezeroq

4.16 1.4.9

release-date 2016-01-08 5:50 P.M PST

release-by Ask Solem

- Fixes compatibility with Linux/macOS instances where the `ctypes` module does not exist.

  Fix contributed by Jared Lewis.

4.17 1.4.8

release-date 2015-12-07 12:25 A.M

release-by Ask Solem

- **`abstract_channel.wait` now accepts a float *timeout* parameter expressed** in seconds

  Contributed by Goir.

4.18 1.4.7

release-date 2015-10-02 05:30 P.M PDT

release-by Ask Solem

- Fixed libSystem error on macOS 10.11 (El Capitan)

  Fix contributed by Eric Wang.

- **`channel.basic_publish` now raises `amqp.exceptions.NotConfirmed` on** `basic.nack`.

- **AMQP timestamps received are now converted from GMT instead of local time** (Issue #67).

- Wheel package installation now supported by both Python 2 and Python3.

  Fix contributed by Rémy Greinhofer.

4.19 1.4.6

release-date 2014-08-11 06:00 P.M UTC

release-by Ask Solem

- Now keeps buffer when socket times out.

  Fix contributed by Artyom Koval.

- Adds `Connection.Transport` attribute that can be used to specify a different transport implementation.

  Contributed by Yury Selivanov.

4.20 1.4.5

release-date 2014-04-15 09:00 P.M UTC

release-by Ask Solem

- Can now deserialize more AMQP types.

  Now handles types `short string`, `short short int`, `short short unsigned int`, `short int`, `short unsigned int`, `long unsigned int`, `long long int`, `long long unsigned int` and `float` which for some reason was missing, even in the original amqplib module.

- SSL: Workaround for Python SSL bug.

  A bug in the python socket library causes `ssl.read/write()` on a closed socket to raise `AttributeError` instead of `IOError`.

  Fix contributed by Craig Jellick.

- `Transport.__del_` now handles errors occurring at late interpreter shutdown (Issue #36).

4.21 1.4.4

release-date 2014-03-03 04:00 P.M UTC

release-by Ask Solem

- SSL transport accidentally disconnected after read timeout.

  Fix contributed by Craig Jellick.

4.22 1.4.3

release-date 2014-02-09 03:00 P.M UTC

release-by Ask Solem

- Fixed bug where more data was requested from the socket than was actually needed.

  Contributed by Ionel Cristian Mărieș.

4.23 1.4.2

release-date 2014-01-23 05:00 P.M UTC

- Heartbeat negotiation would use heartbeat value from server even if heartbeat disabled (Issue #31).

4.24 1.4.1

release-date 2014-01-14 09:30 P.M UTC

release-by Ask Solem

- Fixed error occurring when heartbeats disabled.

4.25 1.4.0

release-date 2014-01-13 03:00 P.M UTC

release-by Ask Solem

- Heartbeat implementation improved (Issue #6).

  The new heartbeat behavior is the same approach as taken by the RabbitMQ java library.

  This also means that clients should preferably call the `heartbeat_tick` method more frequently (like every second) instead of using the old `rate` argument (which is now ignored).

  - Heartbeat interval is negotiated with the server.
  - Some delay is allowed if the heartbeat is late.
  - Monotonic time is used to keep track of the heartbeat instead of relying on the caller to call the checking function at the right time.

  Contributed by Dustin J. Mitchell.

- NoneType is now supported in tables and arrays.

  Contributed by Dominik Fässler.

- SSLTransport: Now handles `ENOENT`.

  Fix contributed by Adrien Guinet.

4.26 1.3.3

release-date 2013-11-11 03:30 P.M UTC

release-by Ask Solem

- SSLTransport: Now keeps read buffer if an exception is raised (Issue #26).

  Fix contributed by Tommie Gannert.

4.27 1.3.2

release-date 2013-10-29 02:00 P.M UTC

release-by Ask Solem

- Message.channel is now a channel object (not the channel id).
- Bug in previous version caused the socket to be flagged as disconnected at EAGAIN/EINTR.

4.28 1.3.1

release-date 2013-10-24 04:00 P.M UTC

release-by Ask Solem

- Now implements Connection.connected (Issue #22).
- Fixed bug where `str(AMQPError)` did not return string.

4.29 1.3.0

release-date 2013-09-04 02:39 P.M UTC

release-by Ask Solem

- Now sets `Message.channel` on delivery (Issue #12)

  amqplib used to make the channel object available as `Message.delivery_info['channel']`, but this was removed in py-amqp. librabbitmq sets `Message.channel`, which is a more reasonable solution in our opinion as that keeps the delivery info intact.

- New option to wait for publish confirmations (Issue #3)

  There is now a new Connection `confirm_publish` that will force any `basic_publish` call to wait for confirmation.

  Enabling publisher confirms like this degrades performance considerably, but can be suitable for some applications and now it's possible by configuration.

- `queue_declare` now returns named tuple of type `basic_declare_ok_t`.

  Supporting fields: `queue`, `message_count`, and `consumer_count`.

- Contents of `Channel.returned_messages` is now named tuples.

  Supporting fields: `reply_code`, `reply_text`, `exchange`, `routing_key`, and `message`.

- Sockets now set to close on exec using the `FD_CLOEXEC` flag.

  Currently only supported on platforms supporting this flag, which does not include Windows.

  Contributed by Tommie Gannert.

4.30 1.2.1

release-date 2013-08-16 05:30 P.M UTC

release-by Ask Solem

- Adds promise type: `amqp.utils.promise()`
- Merges fixes from 1.0.x

4.31 1.2.0

release-date 2012-11-12 04:00 P.M UTC

release-by Ask Solem

- New exception hierarchy:
  - **AMQPError**
    - **ConnectionError**
      - **RecoverableConnectionError**
        - ConsumerCancelled
        - ConnectionForced
        - ResourceError
      - **IrrecoverableConnectionError**
        - ChannelNotOpen
        - FrameError
        - FrameSyntaxError
        - InvalidCommand
        - InvalidPath
        - NotAllowed
        - UnexpectedFrame
        - AMQPNotImplementedError
        - InternalError
    - **ChannelError**
      - **RecoverableChannelError**
        - ContentTooLarge
        - NoConsumers
        - ResourceLocked
      - **IrrecoverableChannelError**
        - AccessRefused
        - NotFound
        - PreconditionFailed

4.32 1.1.0

release-date 2013-11-08 10:36 P.M UTC

release-by Ask Solem

- No longer supports Python 2.5
- Fixed receiving of float table values.
- Now Supports Python 3 and Python 2.6+ in the same source code.
- Python 3 related fixes.

4.33 1.0.13

release-date 2013-07-31 04:00 P.M BST

release-by Ask Solem

- Fixed problems with the SSL transport (Issue #15).

  Fix contributed by Adrien Guinet.

- Small optimizations

4.34 1.0.12

release-date 2013-06-25 02:00 P.M BST

release-by Ask Solem

- Fixed another Python 3 compatibility problem.

4.35 1.0.11

release-date 2013-04-11 06:00 P.M BST

release-by Ask Solem

- Fixed Python 3 incompatibility in `amqp/transport.py`.

4.36 1.0.10

release-date 2013-03-21 03:30 P.M UTC

release-by Ask Solem

- Fixed Python 3 incompatibility in `amqp/serialization.py`. (Issue #11).

4.37 1.0.9

release-date 2013-03-08 10:40 A.M UTC

release-by Ask Solem

- Publisher ack callbacks should now work after typo fix (Issue #9).
- `channel(explicit_id)` will now claim that id from the array of unused channel ids.
- Fixes Jython compatibility.

4.38 1.0.8

release-date 2013-02-08 01:00 P.M UTC

release-by Ask Solem

- Fixed SyntaxError on Python 2.5

4.39 1.0.7

release-date 2013-02-08 01:00 P.M UTC

release-by Ask Solem

- Workaround for bug on some Python 2.5 installations where (2**32) is 0.
- Can now serialize the ARRAY type.

  Contributed by Adam Wentz.
- Fixed tuple format bug in exception (Issue #4).

4.40 1.0.6

release-date 2012-11-29 01:14 P.M UTC

release-by Ask Solem

- `Channel.close` is now ignored if the connection attribute is None.

4.41 1.0.5

release-date 2012-11-21 04:00 P.M UTC

release-by Ask Solem

- `Channel.basic_cancel` is now ignored if the channel was already closed.
- `Channel.events` is now a dict of sets:

```
>>> channel.events['basic_return'].add(on_basic_return)
>>> channel.events['basic_return'].discard(on_basic_return)
```

4.42 1.0.4

release-date 2012-11-13 04:00 P.M UTC

release-by Ask Solem

- Fixes Python 2.5 support

4.43 1.0.3

release-date 2012-11-12 04:00 P.M UTC

release-by Ask Solem

- Now can also handle float in headers/tables when receiving messages.
- Now uses `array.array` to keep track of unused channel ids.
- The `METHOD_NAME_MAP` has been updated for amqp/0.9.1 and Rabbit extensions.
- Removed a bunch of accidentally included images.

4.44 1.0.2

release-date 2012-11-06 05:00 P.M UTC

release-by Ask Solem

- Now supports float values in headers/tables.

4.45 1.0.1

release-date 2012-11-05 01:00 P.M UTC

release-by Ask Solem

- Connection errors no longer includes `AttributeError`.
- Fixed problem with using the SSL transport in a non-blocking context.

  Fix contributed by Mher Movsisyan.

4.46 1.0.0

release-date 2012-11-05 01:00 P.M UTC

release-by Ask Solem

- Channels are now restored on channel error, so that the connection does not have to closed.

4.47 Version 0.9.4

- Adds support for `exchange_bind` and `exchange_unbind`.

  Contributed by Rumyana Neykova

- Fixed bugs in funtests and demo scripts.

  Contributed by Rumyana Neykova

4.48 Version 0.9.3

- Fixed bug that could cause the consumer to crash when reading large message payloads asynchronously.
- Serialization error messages now include the invalid value.

4.49 Version 0.9.2

- Consumer cancel notification support was broken (Issue #1)

  Fix contributed by Andrew Grangaard

4.50 Version 0.9.1

- Supports draining events from multiple channels (`Connection.drain_events`)
- Support for timeouts
- **Support for heartbeats**
  - `Connection.heartbeat_tick(rate=2)` must called at regular intervals (half of the heartbeat value if rate is 2).
  - Or some other scheme by using `Connection.send_heartbeat`.
- **Supports RabbitMQ extensions:**
  - **Consumer Cancel Notifications**
    - by default a cancel results in `ChannelError` being raised
    - but not if a `on_cancel` callback is passed to `basic_consume`.
  - **Publisher confirms**
    - `Channel.confirm_select()` enables publisher confirms.
    - `Channel.events['basic_ack'].append(my_callback)` adds a callback to be called when a message is confirmed. This callback is then called with the signature `(delivery_tag, multiple)`.
- Support for `basic_return`
- **Uses AMQP 0-9-1 instead of 0-8.**
  - `Channel.access_request` and `ticket` arguments to methods **removed**.
  - Supports the `arguments` argument to `basic_consume`.

  - `internal` argument to `exchange_declare` removed.
  - `auto_delete` argument to `exchange_declare` deprecated
  - `insist` argument to `Connection` removed.
  - `Channel.alerts` has been removed.
  - Support for `Channel.basic_recover_async`.
  - `Channel.basic_recover` deprecated.
- **Exceptions renamed to have idiomatic names:**
  - `AMQPException` -> `AMQPError`
  - `AMQPConnectionException` -> ConnectionError‘‘
  - `AMQPChannelException` -> ChannelError‘‘
  - `Connection.known_hosts` removed.
  - `Connection` no longer supports redirects.
  - `exchange` argument to `queue_bind` can now be empty to use the “default exchange”.
- Adds `Connection.is_alive` that tries to detect whether the connection can still be used.
- Adds `Connection.connection_errors` and `.channel_errors`, a list of recoverable errors.
- Exposes the underlying socket as `Connection.sock`.
- Adds `Channel.no_ack_consumers` to keep track of consumer tags that set the no_ack flag.
- Slightly better at error recovery

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

amqp.abstract_channel, 46
amqp.basic_message, 43
amqp.channel, 26
amqp.connection, 9
amqp.exceptions, 44
amqp.five, 53
amqp.method_framing, 47
amqp.platform, 47
amqp.protocol, 47
amqp.sasl, 48
amqp.serialization, 49
amqp.spec, 50
amqp.transport, 46
amqp.utils, 52

Index

A

AbstractChannel (*class in amqp.abstract_channel*), 46
AccessRefused, 45
Ack (*amqp.spec.Basic attribute*), 50
amqp.abstract_channel (*module*), 46
amqp.basic_message (*module*), 43
amqp.channel (*module*), 26
amqp.connection (*module*), 9
amqp.exceptions (*module*), 44
amqp.five (*module*), 53
amqp.method_framing (*module*), 47
amqp.platform (*module*), 47
amqp.protocol (*module*), 47
amqp.sasl (*module*), 48
amqp.serialization (*module*), 49
amqp.spec (*module*), 50
amqp.transport (*module*), 46
amqp.utils (*module*), 52
AMQPDeprecationWarning, 46
AMQPError, 44
AMQPLAIN (*class in amqp.sasl*), 48
AMQPNotImplementedError, 46
append() (*amqp.five.array method*), 57
append() (*amqp.five.UserList method*), 54
args (*amqp.spec.method_t attribute*), 52
array (*class in amqp.five*), 56

B

Basic (*class in amqp.spec*), 50
basic_ack() (*amqp.channel.Channel method*), 27
basic_ack() (*amqp.connection.Connection.Channel method*), 10
basic_cancel() (*amqp.channel.Channel method*), 27
basic_cancel() (*amqp.connection.Connection.Channel method*), 11
basic_consume() (*amqp.channel.Channel method*), 28
basic_consume() (*amqp.connection.Connection.Channel method*), 11
basic_get() (*amqp.channel.Channel method*), 29
basic_get() (*amqp.connection.Connection.Channel method*), 12
basic_publish() (*amqp.channel.Channel method*), 29
basic_publish() (*amqp.connection.Connection.Channel method*), 13
basic_publish_confirm() (*amqp.channel.Channel method*), 30
basic_publish_confirm() (*amqp.connection.Connection.Channel method*), 13
basic_qos() (*amqp.channel.Channel method*), 30
basic_qos() (*amqp.connection.Connection.Channel method*), 13
basic_recover() (*amqp.channel.Channel method*), 31
basic_recover() (*amqp.connection.Connection.Channel method*), 14
basic_recover_async() (*amqp.channel.Channel method*), 31
basic_recover_async() (*amqp.connection.Connection.Channel method*), 14
basic_reject() (*amqp.channel.Channel method*), 31
basic_reject() (*amqp.connection.Connection.Channel method*), 14
basic_return_t (*class in amqp.protocol*), 47
Bind (*amqp.spec.Exchange attribute*), 51
Bind (*amqp.spec.Queue attribute*), 51
BindOk (*amqp.spec.Exchange attribute*), 51
BindOk (*amqp.spec.Queue attribute*), 51
Blocked (*amqp.spec.Connection attribute*), 51
blocking_read() (*amqp.connection.Connection method*), 24
buffer_info() (*amqp.five.array method*), 57
buffer_t (*class in amqp.five*), 61

bytes_if_py2() (*in module amqp.five*), 59
bytes_recv (*amqp.connection.Connection attribute*), 24
bytes_sent (*amqp.connection.Connection attribute*), 24
bytes_t (*in module amqp.five*), 59
bytes_to_str() (*in module amqp.utils*), 52
byteswap() (*amqp.five.array method*), 57

C

Callable (*class in amqp.five*), 55
Cancel (*amqp.spec.Basic attribute*), 50
CancelOk (*amqp.spec.Basic attribute*), 50
Channel (*class in amqp.channel*), 26
Channel (*class in amqp.spec*), 50
channel() (*amqp.connection.Connection method*), 24
channel_errors (*amqp.connection.Connection attribute*), 24
ChannelError, 44
ChannelNotOpen, 45
CLASS_ID (*amqp.basic_message.Message attribute*), 43
CLASS_ID (*amqp.serialization.GenericContent attribute*), 49
CLASS_ID (*amqp.spec.Basic attribute*), 50
CLASS_ID (*amqp.spec.Channel attribute*), 50
CLASS_ID (*amqp.spec.Confirm attribute*), 50
CLASS_ID (*amqp.spec.Connection attribute*), 51
CLASS_ID (*amqp.spec.Exchange attribute*), 51
CLASS_ID (*amqp.spec.Queue attribute*), 51
CLASS_ID (*amqp.spec.Tx attribute*), 52
clear() (*amqp.five.UserList method*), 54
client_heartbeat (*amqp.connection.Connection attribute*), 24
Close (*amqp.spec.Channel attribute*), 50
Close (*amqp.spec.Connection attribute*), 51
close() (*amqp.abstract_channel.AbstractChannel method*), 46
close() (*amqp.channel.Channel method*), 32
close() (*amqp.connection.Connection method*), 24
close() (*amqp.connection.Connection.Channel method*), 15
close() (*amqp.five.StringIO method*), 60
closed (*amqp.five.StringIO attribute*), 60
CloseOk (*amqp.spec.Channel attribute*), 50
CloseOk (*amqp.spec.Connection attribute*), 51
code (*amqp.exceptions.AccessRefused attribute*), 45
code (*amqp.exceptions.AMQPError attribute*), 44
code (*amqp.exceptions.AMQPNotImplementedError attribute*), 46
code (*amqp.exceptions.ChannelNotOpen attribute*), 45
code (*amqp.exceptions.ConnectionForced attribute*), 45
code (*amqp.exceptions.ContentTooLarge attribute*), 44
code (*amqp.exceptions.FrameError attribute*), 45
code (*amqp.exceptions.FrameSyntaxError attribute*), 45
code (*amqp.exceptions.InternalError attribute*), 46
code (*amqp.exceptions.InvalidCommand attribute*), 45
code (*amqp.exceptions.InvalidPath attribute*), 45
code (*amqp.exceptions.NoConsumers attribute*), 44
code (*amqp.exceptions.NotAllowed attribute*), 46
code (*amqp.exceptions.NotFound attribute*), 45
code (*amqp.exceptions.PreconditionFailed attribute*), 45
code (*amqp.exceptions.ResourceError attribute*), 46
code (*amqp.exceptions.ResourceLocked attribute*), 45
code (*amqp.exceptions.UnexpectedFrame attribute*), 45
collect() (*amqp.channel.Channel method*), 33
collect() (*amqp.connection.Connection method*), 25
collect() (*amqp.connection.Connection.Channel method*), 16
Commit (*amqp.spec.Tx attribute*), 52
CommitOk (*amqp.spec.Tx attribute*), 52
Confirm (*class in amqp.spec*), 50
confirm_select() (*amqp.channel.Channel method*), 33
confirm_select() (*amqp.connection.Connection.Channel method*), 16
connect() (*amqp.connection.Connection method*), 25
connected (*amqp.connection.Connection attribute*), 25
Connection (*class in amqp.connection*), 9
Connection (*class in amqp.spec*), 50
Connection.Channel (*class in amqp.connection*), 10
connection_errors (*amqp.connection.Connection attribute*), 25
ConnectionError, 44
ConnectionForced, 45
Consume (*amqp.spec.Basic attribute*), 50
ConsumeOk (*amqp.spec.Basic attribute*), 50
consumer_count (*amqp.protocol.queue_declare_ok_t attribute*), 48
ConsumerCancelled, 44
content (*amqp.spec.method_t attribute*), 52
ContentTooLarge, 44
copy() (*amqp.five.Counter method*), 53
copy() (*amqp.five.UserDict method*), 55
copy() (*amqp.five.UserList method*), 54
coro() (*in module amqp.utils*), 52
count() (*amqp.five.array method*), 57
count() (*amqp.five.range method*), 59
count() (*amqp.five.UserList method*), 55
Counter (*class in amqp.five*), 53

D

Declare (*amqp.spec.Exchange attribute*), 51
Declare (*amqp.spec.Queue attribute*), 51
DeclareOk (*amqp.spec.Exchange attribute*), 51
DeclareOk (*amqp.spec.Queue attribute*), 51

decode_properties_basic() (*in module amqp.serialization*), 49
Delete (*amqp.spec.Exchange attribute*), 51
Delete (*amqp.spec.Queue attribute*), 51
DeleteOk (*amqp.spec.Exchange attribute*), 51
DeleteOk (*amqp.spec.Queue attribute*), 51
Deliver (*amqp.spec.Basic attribute*), 50
delivery_info (*amqp.basic_message.Message attribute*), 43
delivery_tag (*amqp.basic_message.Message attribute*), 43
dispatch_method() (*amqp.abstract_channel.AbstractChannel method*), 46
drain_events() (*amqp.connection.Connection method*), 25
dumps() (*in module amqp.serialization*), 49

E

elements() (*amqp.five.Counter method*), 53
emit() (*amqp.utils.NullHandler method*), 52
Empty, 56
empty() (*amqp.five.Queue method*), 55
exchange (*amqp.protocol.basic_return_t attribute*), 47
Exchange (*class in amqp.spec*), 51
exchange_bind() (*amqp.channel.Channel method*), 33
exchange_bind() (*amqp.connection.Connection.Channel method*), 16
exchange_declare() (*amqp.channel.Channel method*), 34
exchange_declare() (*amqp.connection.Connection.Channel method*), 16
exchange_delete() (*amqp.channel.Channel method*), 35
exchange_delete() (*amqp.connection.Connection.Channel method*), 18
exchange_unbind() (*amqp.channel.Channel method*), 36
exchange_unbind() (*amqp.connection.Connection.Channel method*), 18
extend() (*amqp.five.array method*), 57
extend() (*amqp.five.UserList method*), 55
EXTERNAL (*class in amqp.sasl*), 48

F

Flow (*amqp.spec.Channel attribute*), 50
flow() (*amqp.channel.Channel method*), 36
flow() (*amqp.connection.Connection.Channel method*), 19
FlowOk (*amqp.spec.Channel attribute*), 50
format_d() (*in module amqp.five*), 61
frame_handler() (*in module amqp.method_framing*), 47
frame_writer (*amqp.connection.Connection attribute*), 25
frame_writer() (*in module amqp.method_framing*), 47
FrameError, 45
FrameSyntaxError, 45
frombytes() (*amqp.five.array method*), 57
fromfile() (*amqp.five.array method*), 57
fromkeys() (*amqp.five.Counter class method*), 54
fromkeys() (*amqp.five.UserDict class method*), 55
fromlist() (*amqp.five.array method*), 57
fromstring() (*amqp.five.array method*), 57
fromunicode() (*amqp.five.array method*), 58
Full, 56
full() (*amqp.five.Queue method*), 55

G

GenericContent (*class in amqp.serialization*), 49
Get (*amqp.spec.Basic attribute*), 50
get() (*amqp.five.Mapping method*), 55
get() (*amqp.five.Queue method*), 56
get_errno() (*in module amqp.utils*), 52
get_logger() (*in module amqp.utils*), 52
get_nowait() (*amqp.five.Queue method*), 56
GetEmpty (*amqp.spec.Basic attribute*), 50
getfullargspec() (*in module amqp.five*), 61
GetOk (*amqp.spec.Basic attribute*), 50
getvalue() (*amqp.five.StringIO method*), 60
GSSAPI (*in module amqp.sasl*), 48

H

headers (*amqp.basic_message.Message attribute*), 44
heartbeat (*amqp.connection.Connection attribute*), 25
heartbeat_tick() (*amqp.connection.Connection method*), 25

I

inbound_body() (*amqp.serialization.GenericContent method*), 49
inbound_header() (*amqp.serialization.GenericContent method*), 49
index() (*amqp.five.array method*), 58
index() (*amqp.five.range method*), 59
index() (*amqp.five.UserList method*), 55
insert() (*amqp.five.array method*), 58
insert() (*amqp.five.UserList method*), 55
InternalError, 46
InvalidCommand, 45
InvalidPath, 45
IrrecoverableChannelError, 44

IrrecoverableConnectionError, 44
is_alive() (amqp.connection.Connection method), 25
items() (amqp.five.Mapping method), 55
items() (in module amqp.five), 59
itemsize (amqp.five.array attribute), 58
Iterable (class in amqp.five), 55

J

join() (amqp.five.Queue method), 56

K

keys() (amqp.five.Mapping method), 55
keys() (in module amqp.five), 59

L

last_heartbeat_received (amqp.connection.Connection attribute), 25
last_heartbeat_sent (amqp.connection.Connection attribute), 25
library_properties (amqp.connection.Connection attribute), 25
LifoQueue (class in amqp.five), 56
line_buffering (amqp.five.StringIO attribute), 60
loads() (in module amqp.serialization), 49
long_t (in module amqp.five), 59

M

map (class in amqp.five), 58
Mapping (class in amqp.five), 55
mechanism (amqp.sasl.AMQPLAIN attribute), 48
mechanism (amqp.sasl.EXTERNAL attribute), 48
mechanism (amqp.sasl.PLAIN attribute), 48
mechanism (amqp.sasl.RAW attribute), 49
mechanism (amqp.sasl.SASL attribute), 49
message (amqp.protocol.basic_return_t attribute), 47
Message (class in amqp.basic_message), 43
message_count (amqp.protocol.queue_declare_ok_t attribute), 48
method (amqp.exceptions.AMQPError attribute), 44
method() (in module amqp.spec), 52
method_sig (amqp.spec.method_t attribute), 52
method_t (class in amqp.spec), 52
module_name_t (in module amqp.five), 59
monotonic() (in module amqp.five), 61
most_common() (amqp.five.Counter method), 54

N

Nack (amqp.spec.Basic attribute), 50
negotiate_capabilities (amqp.connection.Connection attribute), 26
newlines (amqp.five.StringIO attribute), 60
nextfun() (in module amqp.five), 59
NoConsumers, 44
NotAllowed, 46
NotFound, 45
NullHandler (class in amqp.utils), 52

O

on_inbound_frame (amqp.connection.Connection attribute), 26
on_inbound_method() (amqp.connection.Connection method), 26
Open (amqp.spec.Channel attribute), 50
Open (amqp.spec.Connection attribute), 51
open() (amqp.channel.Channel method), 37
open() (amqp.connection.Connection.Channel method), 19
OpenOk (amqp.spec.Channel attribute), 50
OpenOk (amqp.spec.Connection attribute), 51

P

pack() (in module amqp.platform), 47
pack_into() (in module amqp.platform), 47
PLAIN (class in amqp.sasl), 48
pop() (amqp.five.array method), 58
pop() (amqp.five.UserList method), 55
PreconditionFailed, 45
prev_recv (amqp.connection.Connection attribute), 26
prev_sent (amqp.connection.Connection attribute), 26
PROPERTIES (amqp.basic_message.Message attribute), 43
PROPERTIES (amqp.serialization.GenericContent attribute), 49
Publish (amqp.spec.Basic attribute), 50
Purge (amqp.spec.Queue attribute), 51
PurgeOk (amqp.spec.Queue attribute), 51
put() (amqp.five.Queue method), 56
put_nowait() (amqp.five.Queue method), 56
python_2_unicode_compatible() (in module amqp.five), 61

Q

Qos (amqp.spec.Basic attribute), 50
QosOk (amqp.spec.Basic attribute), 50
qsize() (amqp.five.Queue method), 56
queue (amqp.protocol.queue_declare_ok_t attribute), 48
Queue (class in amqp.five), 55
Queue (class in amqp.spec), 51

queue_bind() (*amqp.channel.Channel method*), 37
queue_bind() (*amqp.connection.Connection.Channel method*), 19
queue_declare() (*amqp.channel.Channel method*), 38
queue_declare() (*amqp.connection.Connection.Channel method*), 20
queue_declare_ok_t (*class in amqp.protocol*), 48
queue_delete() (*amqp.channel.Channel method*), 40
queue_delete() (*amqp.connection.Connection.Channel method*), 22
queue_purge() (*amqp.channel.Channel method*), 41
queue_purge() (*amqp.connection.Connection.Channel method*), 23
queue_unbind() (*amqp.channel.Channel method*), 42
queue_unbind() (*amqp.connection.Connection.Channel method*), 23

R

range (*class in amqp.five*), 59
RAW (*class in amqp.sasl*), 48
read() (*amqp.five.StringIO method*), 60
readable() (*amqp.five.StringIO method*), 60
readline() (*amqp.five.StringIO method*), 60
Recover (*amqp.spec.Basic attribute*), 50
recoverable_channel_errors (*amqp.connection.Connection attribute*), 26
recoverable_connection_errors (*amqp.connection.Connection attribute*), 26
RecoverableChannelError, 44
RecoverableConnectionError, 44
RecoverAsync (*amqp.spec.Basic attribute*), 50
RecoverOk (*amqp.spec.Basic attribute*), 50
Reject (*amqp.spec.Basic attribute*), 50
reload() (*in module amqp.five*), 54
remove() (*amqp.five.array method*), 58
remove() (*amqp.five.UserList method*), 55
reply_code (*amqp.protocol.basic_return_t attribute*), 48
reply_text (*amqp.protocol.basic_return_t attribute*), 48
reraise() (*in module amqp.five*), 59
ResourceError, 45
ResourceLocked, 45
Return (*amqp.spec.Basic attribute*), 50
reverse() (*amqp.five.array method*), 58
reverse() (*amqp.five.UserList method*), 55
Rollback (*amqp.spec.Tx attribute*), 52
RollbackOk (*amqp.spec.Tx attribute*), 52
routing_key (*amqp.protocol.basic_return_t attribute*), 48

S

SASL (*class in amqp.sasl*), 49
Secure (*amqp.spec.Connection attribute*), 51
SecureOk (*amqp.spec.Connection attribute*), 51
seek() (*amqp.five.StringIO method*), 60
seekable() (*amqp.five.StringIO method*), 60
Select (*amqp.spec.Confirm attribute*), 50
Select (*amqp.spec.Tx attribute*), 52
SelectOk (*amqp.spec.Confirm attribute*), 50
SelectOk (*amqp.spec.Tx attribute*), 52
send_heartbeat() (*amqp.connection.Connection method*), 26
send_method() (*amqp.abstract_channel.AbstractChannel method*), 46
server_capabilities (*amqp.connection.Connection attribute*), 26
server_heartbeat (*amqp.connection.Connection attribute*), 26
set_cloexec() (*in module amqp.utils*), 52
sock (*amqp.connection.Connection attribute*), 26
sort() (*amqp.five.UserList method*), 55
SSLTransport (*class in amqp.transport*), 46
start (*amqp.five.range attribute*), 59
Start (*amqp.spec.Connection attribute*), 51
start() (*amqp.sasl.AMQPLAIN method*), 48
start() (*amqp.sasl.EXTERNAL method*), 48
start() (*amqp.sasl.PLAIN method*), 48
start() (*amqp.sasl.RAW method*), 49
start() (*amqp.sasl.SASL method*), 49
StartOk (*amqp.spec.Connection attribute*), 51
step (*amqp.five.range attribute*), 59
stop (*amqp.five.range attribute*), 59
str_to_bytes() (*in module amqp.utils*), 52
string (*in module amqp.five*), 59
string_t (*in module amqp.five*), 59
StringIO (*class in amqp.five*), 60
subtract() (*amqp.five.Counter method*), 54

T

task_done() (*amqp.five.Queue method*), 56
TCPTransport (*class in amqp.transport*), 46
tell() (*amqp.five.StringIO method*), 61
text_t (*in module amqp.five*), 59
then() (*amqp.channel.Channel method*), 42
then() (*amqp.connection.Connection method*), 26
then() (*amqp.connection.Connection.Channel method*), 24
to_host_port() (*in module amqp.transport*), 47
tobytes() (*amqp.five.array method*), 58
tofile() (*amqp.five.array method*), 58

tolist() (*amqp.five.array method*), 58
tostring() (*amqp.five.array method*), 58
tounicode() (*amqp.five.array method*), 58
transport (*amqp.connection.Connection attribute*), 26
Transport() (*amqp.connection.Connection method*), 24
Transport() (*in module amqp.transport*), 46
truncate() (*amqp.five.StringIO method*), 61
Tune (*amqp.spec.Connection attribute*), 51
TuneOk (*amqp.spec.Connection attribute*), 51
Tx (*class in amqp.spec*), 52
tx_commit() (*amqp.channel.Channel method*), 42
tx_commit() (*amqp.connection.Connection.Channel method*), 24
tx_rollback() (*amqp.channel.Channel method*), 42
tx_rollback() (*amqp.connection.Connection.Channel method*), 24
tx_select() (*amqp.channel.Channel method*), 42
tx_select() (*amqp.connection.Connection.Channel method*), 24
typecode (*amqp.five.array attribute*), 58

U

Unbind (*amqp.spec.Exchange attribute*), 51
Unbind (*amqp.spec.Queue attribute*), 51
UnbindOk (*amqp.spec.Exchange attribute*), 51
UnbindOk (*amqp.spec.Queue attribute*), 52
Unblocked (*amqp.spec.Connection attribute*), 51
UnexpectedFrame, 45
unpack() (*in module amqp.platform*), 47
unpack_from() (*in module amqp.platform*), 47
update() (*amqp.five.Counter method*), 54
UserDict (*class in amqp.five*), 55
UserList (*class in amqp.five*), 54

V

values() (*amqp.five.Mapping method*), 55
values() (*in module amqp.five*), 59

W

wait() (*amqp.abstract_channel.AbstractChannel method*), 46
WhateverIO (*class in amqp.five*), 60
with_metaclass() (*in module amqp.five*), 60
writable() (*amqp.five.StringIO method*), 61
write() (*amqp.five.StringIO method*), 61
write() (*amqp.five.WhateverIO method*), 60

Z

zip (*class in amqp.five*), 59
zip_longest (*class in amqp.five*), 58