amqpy

Release 0.13.1

May 22, 2016

Contents

API documentation:

CHAPTER 1

amqpy.connection module

AMQP Connections

class amqpy.connection.**Connection**(*amqpy.abstract_channel.AbstractChannel*)
Bases: amqpy.abstract_channel.AbstractChannel

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

connected
@property

Check if connection is connected

Returns True if connected, else False

Return type bool

server_capabilities
@property

Get server capabilities

These properties are set only after successfully connecting.

Returns server capabilities

Return type dict

sock
@property

Access underlying TCP socket

Returns socket

Return type socket.socket

channels = None
Map of *{channel_id: Channel}* for all active channels

Type dict[int, Channel]

transport = None

Type amqpy.transport.Transport

__init__(*host='localhost'*, *port=5672*, *ssl=None*, *connect_timeout=None*, *userid='guest'*, *password='guest'*, *login_method='AMQPLAIN'*, *virtual_host='/'*, *locale='en_US'*, *channel_max=65535*, *frame_max=131072*, *heartbeat=0*, *client_properties=None*, *on_blocked=None*, *on_unblocked=None*)

Create a connection to the specified host

If you are using SSL, make sure the correct port number is specified (usually 5671), as the default of 5672 is for non-SSL connections.

Parameters

- **host** (*str*) – host
- **port** (*int*) – port
- **ssl** (*dict or None*) – dict of SSL options passed to `ssl.wrap_socket()`, None to disable SSL
- **connect_timeout** (*float or None*) – connect timeout
- **userid** (*str*) – username
- **password** (*str*) – password
- **login_method** (*str*) – login method (this is server-specific); default is for RabbitMQ
- **virtual_host** (*str*) – virtual host
- **locale** (*str*) – locale
- **channel_max** (*int*) – maximum number of channels
- **frame_max** (*int*) – maximum frame payload size in bytes
- **heartbeat** (*float*) – heartbeat interval in seconds, 0 disables heartbeat
- **client_properties** (*dict or None*) – dict of client properties
- **on_blocked** (*Callable or None*) – callback on connection blocked
- **on_unblocked** (*Callable or None*) – callback on connection unblocked

channel(*channel_id=None*) → amqpy.channel.Channel

Create a new channel, or fetch the channel associated with *channel_id* if specified

Parameters **channel_id** (*int or None*) – channel ID number

Returns Channel

Return type *amqpy.channel.Channel*

close(*reply_code=0*, *reply_text=''*, *method_type=method_t(class_id=0, method_id=0)*) → None

Close connection to the server

This method performs a connection close handshake with the server, then closes the underlying connection.

If this connection close is due to a client error, the client may provide a *reply_code*, *reply_text*, and *method_type* to indicate to the server the reason for closing the connection.

Parameters

- **reply_code** (*int*) – the reply code
- **reply_text** (*str*) – localized reply text
- **method_type** (`amqpy.spec.method_t`) – if close is triggered by a failing method, this is the method that caused it

connect() → None
Connect using saved connection parameters

This method does not need to be called explicitly; it is called by the constructor during initialization.

Note: reconnecting invalidates all declarations (channels, queues, consumers, delivery tags, etc.).

drain_events(*timeout=None*) → None
Wait for an event on all channels

This method should be called after creating consumers in order to receive delivered messages and execute consumer callbacks.

Parameters **timeout** (*float or None*) – maximum allowed time to wait for an event

Raises ***amqpy.exceptions.Timeout*** – if the operation times out

is_alive() → bool
Check if connection is alive

This method is the primary way to check if the connection is alive.

Side effects: This method may send a heartbeat as a last resort to check if the connection is alive.

Returns True if connection is alive, else False

Return type bool

loop(*timeout=None*) → None
Call *drain_events()* continuously

•Does not raise Timeout exceptions if a timeout occurs

Parameters **timeout** (*float or None*) – maximum allowed time to wait for an event

send_heartbeat() → None
Send a heartbeat to the server

CHAPTER 2

amqpy.channel module

AMQP Channels

class amqpy.channel.**Channel**(*amqpy.abstract_channel.AbstractChannel*)
Bases: amqpy.abstract_channel.AbstractChannel

The channel class provides methods for a client to establish and operate an AMQP channel. All public members are fully thread-safe.

CH_MODE_CONFIRM = 2
Publisher confirm mode (RabbitMQ extension)

CH_MODE_NONE = 0
Default channel mode

CH_MODE_TX = 1
Transaction mode

active = None
Current channel active state (flow control)

Type bool

is_open = None
Current channel open/closed state

Type bool

mode = None
Channel mode state (default, transactional, publisher confirm)

Type int

returned_messages = None
Returned messages that the server was unable to deliver

Type queue.Queue

__init__(*connection*, *channel_id=None*, *auto_decode=True*)
Create a channel bound to a connection and using the specified numeric channel_id, and open on the server

If *auto_decode* is enabled (default), incoming Message bodies will be automatically decoded to *str* if possible.

Parameters

- **connection** (amqpy.connection.Connection) – the channel's associated Connection

- **channel_id** (*int or None*) – the channel's assigned channel ID
- **auto_decode** (*bool*) – enable auto decoding of message bodies

basic_ack(*delivery_tag*, *multiple=False*) → None
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.

•The delivery tag is valid only within the same channel that the message was received.

•Set *delivery_tag* to *0* and *multiple* to *True* to acknowledge all outstanding messages.

•If the *delivery_tag* is invalid, the server must raise a channel exception.

Parameters

- **delivery_tag** (*int*) – server-assigned delivery tag; 0 means "all messages received so far"
- **multiple** (*bool*) – if set, the *delivery_tag* is treated as "all messages up to and including"

basic_cancel(*consumer_tag*, *nowait=False*) → None
End a queue consumer

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer. The client may receive an arbitrary number of messages in between sending the cancel method and receiving the cancel-ok reply.

•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** (*str*) – consumer tag, valid only within the current connection and channel
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply

basic_consume(*queue=''*, *consumer_tag=''*, *no_local=False*, *no_ack=False*, *exclusive=False*, *nowait=False*, *callback=None*, *arguments=None*, *on_cancel=None*) → str
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.

•The *consumer_tag* is local to a connection, so two clients can use the same consumer tags. But on the same connection, the *consumer_tag* must be unique, or the server must raise a 530 NOT ALLOWED connection exception.

•If *no_ack* is set, the server automatically acknowledges each message on behalf of the client.

•If *exclusive* is set, the client asks for this consumer to have exclusive access to the queue. If the server cannot grant exclusive access to the queue because there are other consumers active, it must raise a 403 ACCESS REFUSED channel exception.

•*callback* must be a *Callable(message)* which is called for each messaged delivered by the broker. If no callback is specified, messages are quietly discarded; *no_ack* should probably be set to True in that case.

Parameters

- **queue** (*str*) – name of queue; if None, refers to last declared queue for this channel
- **consumer_tag** (*str*) – consumer tag, local to the connection
- **no_local** (*bool*) – if True: do not deliver own messages
- **no_ack** (*bool*) – server will not expect an ack for each message
- **exclusive** (*bool*) – request exclusive access
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply
- **callback** (*Callable*) – a callback callable(message) for each delivered message
- **arguments** (*dict*) – AMQP method arguments
- **on_cancel** (*Callable*) – a callback callable

Returns consumer tag

Return type str

basic_get(*queue=''*, *no_ack=False*) → amqpy.message.Message or None
Directly get a message from the *queue*

This method is non-blocking. If no messages are available on the queue, *None* is returned.

Parameters

- **queue** (*str*) – queue name; leave blank to refer to last declared queue for the channel
- **no_ack** (*bool*) – if enabled, the server automatically acknowledges the message

Returns message, or None if no messages are available on the queue

Return type amqpy.message.Message or None

basic_publish(*msg*, *exchange=''*, *routing_key=''*, *mandatory=False*, *immediate=False*) → None
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.

If publisher confirms are enabled, this method will automatically wait to receive an "ack" from the server.

Note: Returned messages are sent back from the server and loaded into the *returned_messages* queue of the channel that sent them. In order to receive all returned messages, call *loop(0)* on the connection object before checking the channel's *returned_messages* queue.

Parameters

- **msg** (*amqpy.Message*) – message
- **exchange** (*str*) – exchange name, empty string means default exchange
- **routing_key** (*str*) – routing key
- **mandatory** (*bool*) – True: deliver to at least one queue, or return it; False: drop the unroutable message
- **immediate** (*bool*) – request immediate delivery

basic_qos(*prefetch_size=0*, *prefetch_count=0*, *a_global=False*) → None
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.

- 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.
- The server must ignore *prefetch_size* 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.
- The *prefetch_count* 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.
- The server may send less data in advance than allowed by the client's specified prefetch windows but it must not send more.

Parameters

- **prefetch_size** (*int*) – prefetch window in octets
- **prefetch_count** (*int*) – prefetch window in messages
- **a_global** (*bool*) – apply to entire connection (default is for current channel only)

basic_recover(*requeue=False*) → None
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.

- The server MUST set the redelivered flag on all messages that are resent.
- The server MUST raise a channel exception if this is called on a transacted channel.

Parameters **requeue** (*bool*) – if set, the server will attempt to requeue the message, potentially then delivering it to a different subscriber

basic_recover_async(*requeue=False*) → None
Redeliver unacknowledged messages (async)

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.

- The server MUST set the redelivered flag on all messages that are resent.
- The server MUST raise a channel exception if this is called on a transacted channel.

Parameters **requeue** (*bool*) – if set, the server will attempt to requeue the message, potentially then delivering it to a different subscriber

basic_reject(*delivery_tag*, *requeue*) → None
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.

- 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).
- The server SHOULD interpret this method as meaning that the client is unable to process the message at this time.
- 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.
- 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.

Parameters

- **delivery_tag** (*int*) – server-assigned channel-specific delivery tag
- **requeue** (*bool*) – True: requeue the message; False: discard the message

close(*reply_code=0*, *reply_text=''*, *method_type=method_t(class_id=0, method_id=0)*) → None
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.

Parameters

- **reply_code** (*int*) – the reply code
- **reply_text** (*str*) – localized reply text
- **method_type** (amqpy.spec.method_t) – if close is triggered by a failing method, this is the method that caused it

confirm_select(*nowait=False*) → None
Enable publisher confirms for this channel (RabbitMQ extension)

The channel must not be in transactional mode. If it is, the server raises a PreconditionFailed exception and closes the channel. Note that amqpy will automatically reopen the channel, at which point this method can be called again successfully.

Parameters **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply

Raises **PreconditionFailed** – if the channel is in transactional mode

exchange_bind(*dest_exch*, *source_exch=''*, *routing_key=''*, *nowait=False*, *arguments=None*) → None
Bind an exchange to an exchange

- Both the *dest_exch* and *source_exch* must already exist. Blank exchange names mean the default exchange.

•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.

•A server MUST allow cycles of exchange bindings to be created including allowing an exchange to be bound to itself.

•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

- **dest_exch** (*str*) – name of destination exchange to bind
- **source_exch** (*str*) – name of source exchange to bind
- **routing_key** (*str*) – routing key for the binding (note: not all exchanges use a routing key)
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply
- **arguments** (*dict*) – binding arguments, specific to the exchange class

exchange_declare(*exchange*, *exch_type*, *passive=False*, *durable=False*, *auto_delete=True*, *nowait=False*, *arguments=None*) → None

Declare exchange, create if needed

•Exchanges cannot be redeclared with different types. The client MUST not attempt to redeclare an existing exchange with a different type than used in the original Exchange.Declare method.

•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.

•The server must ignore the *durable* field if the exchange already exists.

•The server must ignore the *auto_delete* field if the exchange already exists.

•If *nowait* is enabled and the server could not complete the method, it will raise a channel or connection exception.

•*arguments* is ignored if passive is True.

Parameters

- **exchange** (*str*) – exchange name
- **exch_type** (*str*) – exchange type (direct, fanout, etc.)
- **passive** (*bool*) – do not create exchange; client can use this to check whether an exchange exists
- **durable** (*bool*) – mark exchange as durable (remain active after server restarts)
- **auto_delete** (*bool*) – auto-delete exchange when all queues have finished using it
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply
- **arguments** (*dict*) – exchange declare arguments

Raises

- **AccessRefused** – if attempting to declare an exchange with a reserved name (amq.*)
- **NotFound** – if *passive* is enabled and the exchange does not exist

Returns None

exchange_delete(*exchange*, *if_unused=False*, *nowait=False*) → None
Delete an exchange

This method deletes an exchange.

•If the exchange does not exist, the server must raise a channel exception. When an exchange is deleted, all queue bindings on the exchange are cancelled.

•If *if_unused* is set, and the exchange has queue bindings, the server must raise a channel exception.

Parameters

- **exchange** (*str*) – exchange name
- **if_unused** (*bool*) – delete only if unused (has no queue bindings)
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply

Raises

- **NotFound** – if exchange with *exchange* does not exist
- **PreconditionFailed** – if attempting to delete a queue with bindings and *if_unused* is set

Returns None

exchange_unbind(*dest_exch*, *source_exch=''*, *routing_key=''*, *nowait=False*, *arguments=None*) → None
Unbind an exchange from an exchange

•**If the unbind fails, the server must raise a connection exception. The server must not** attempt to unbind an exchange that does not exist from an exchange.

•Blank exchange names mean the default exchange.

Parameters

- **dest_exch** (*str*) – destination exchange name
- **source_exch** (*str*) – source exchange name
- **routing_key** (*str*) – routing key to unbind
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply
- **arguments** (*dict*) – binding arguments, specific to the exchange class

flow(*active*) → None
Enable/disable flow from peer

This method asks the peer to pause or restart the flow of content data. This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process. Note that this method is not intended for window control The peer that receives a request to stop sending content should finish sending the current content, if any, and then wait until it receives a Flow restart method.

Parameters **active** (*bool*) – True: peer starts sending content frames; False: peer stops sending content frames

queue_bind(*queue*, *exchange=''*, *routing_key=''*, *nowait=False*, *arguments=None*) → None
Bind queue to an exchange

This method binds a queue to an exchange. Until a queue is bound it will not receive any messages. In a classic messaging model, store-and-forward queues are bound to a dest exchange and subscription queues are bound to a dest_wild exchange.

•The server must allow and ignore duplicate bindings without treating these as an error.

•If a bind fails, the server must raise a connection exception.

•The server must not allow a durable queue to bind to a transient exchange. If a client attempts this, the server must raise a channel exception.

•The server should support at least 4 bindings per queue, and ideally, impose no limit except as defined by available resources.

•If the client did not previously declare a queue, and the *queue* is empty, the server must raise a connection exception with reply code 530 (not allowed).

•If *queue* does not exist, the server must raise a channel exception with reply code 404 (not found).

•If *exchange* does not exist, the server must raise a channel exception with reply code 404 (not found).

Parameters

- **queue** (*str*) – name of queue to bind; blank refers to the last declared queue for this channel
- **exchange** (*str*) – name of exchange to bind to
- **routing_key** (*str*) – routing key for the binding
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply
- **arguments** (*dict*) – binding arguments, specific to the exchange class

queue_declare(*queue=''*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=True*, *nowait=False*, *arguments=None*) → queue_declare_ok_t or None
Declare queue, create if needed

This method creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue. A tuple containing the queue name, message count, and consumer count is returned, which is essential for declaring automatically named queues.

•If *passive* is specified, the server state is not modified (a queue will not be declared), and the server only checks if the specified queue exists and returns its properties. If the queue does not exist, the server must raise a 404 NOT FOUND channel exception.

•The server must create a default binding for a newly-created queue to the default exchange, which is an exchange of type 'direct'.

•Queue names starting with 'amq.' are reserved for use by the server. If an attempt is made to declare a queue with such a name, and the *passive* flag is disabled, the server must raise a 403 ACCESS REFUSED connection exception.

•The server must raise a 405 RESOURCE LOCKED channel exception if an attempt is made to access a queue declared as exclusive by another open connection.

•The server must ignore the *auto_delete* flag if the queue already exists.

RabbitMQ supports the following useful additional arguments:

•**x-max-length (int): maximum queue size**

– Queue length is a measure that takes into account ready messages, ignoring unacknowledged messages and message size. Messages will be dropped or dead-lettered from the front of the queue to make room for new messages once the limit is reached.

Parameters

- **queue** (*str*) – queue name; leave blank to let the server generate a name automatically
- **passive** (*bool*) – do not create queue; client can use this to check whether a queue exists
- **durable** (*bool*) – mark as durable (remain active after server restarts)
- **exclusive** (*bool*) – mark as exclusive (can only be consumed from by this connection); implies *auto_delete*
- **auto_delete** (*bool*) – auto-delete queue when all consumers have finished using it
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply
- **arguments** (*dict*) – exchange declare arguments

Raises

- **NotFound** – if *passive* is enabled and the queue does not exist
- **AccessRefused** – if an attempt is made to declare a queue with a reserved name
- **ResourceLocked** – if an attempt is made to access an exclusive queue declared by another open connection

Returns queue_declare_ok_t(queue, message_count, consumer_count), or None if *nowait*

Return type queue_declare_ok_t or None

queue_delete(*queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*) → int

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.

Parameters

- **queue** (*str*) – name of queue to delete, empty string refers to last declared queue on this channel
- **if_unused** (*bool*) – delete only if unused (has no consumers); raise a channel exception otherwise
- **if_empty** (*bool*) – delete only if empty; raise a channel exception otherwise
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply

Raises

- **NotFound** – if *queue* does not exist
- **PreconditionFailed** – if *if_unused* or *if_empty* conditions are not met

Returns number of messages deleted

Return type int

queue_purge(*queue=''*, *nowait=False*) → int or None
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.

•On transacted channels the server MUST not purge messages that have already been sent to a client but not yet acknowledged.

•If nowait is False, this method returns a message count.

Parameters

- **queue** (*str*) – queue name to purge; leave blank to refer to last declared queue for this channel
- **nowait** (*bool*) – if set, the server will not respond to the method and the client should not wait for a reply

Returns message count (if nowait is False)

Return type int or None

queue_unbind(*queue*, *exchange*, *routing_key=''*, *nowait=False*, *arguments=None*) → None
Unbind a queue from an exchange

This method unbinds a queue from an exchange.

•If a unbind fails, the server MUST raise a connection exception.

•The client must not attempt to unbind a queue that does not exist.

•The client must not attempt to unbind a queue from an exchange that does not exist.

Parameters

- **queue** (*str*) – name of queue to unbind, leave blank to refer to the last declared queue on this channel
- **exchange** (*str*) – name of exchange to unbind, leave blank to refer to default exchange
- **routing_key** (*str*) – routing key of binding
- **arguments** (*dict*) – binding arguments, specific to the exchange class

tx_commit() → None
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() → None
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() → None
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.

The channel must not be in publish acknowledge mode. If it is, the server raises a `PreconditionFailed` exception and closes the channel. Note that amqpy will automatically reopen the channel, at which point this method can be called again successfully.

Raises **`PreconditionFailed`** – if the channel is in publish acknowledge mode

CHAPTER 3

amqpy.message module

Messages for AMQP

class amqpy.message.**Message**(*amqpy.message.GenericContent*)
Bases: amqpy.message.GenericContent

A Message for use with the *Channel.basic_** methods

application_headers
@property

Get application headers

Returns application headers

Return type dict

body
Message body (bytes or str or unicode)

channel
Associated channel, set after receiving a message (amqpy.channel.Channel)

delivery_info
Delivery info, set after receiving a message (dict)

delivery_tag
@property

Get delivery tag

Returns delivery tag

Return type int

__init__(*body=''*, *channel=None*, ***properties*)
If *body* is a *str*, then *content_encoding* will automatically be set to 'UTF-8', unless explicitly specified.

Example:

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

Parameters

- **body** (*bytes or str or unicode*) – message body
- **channel** (amqpy.channel.Channel) – associated channel
- **properties** –

- content_type (shortstr): MIME content type
- content_encoding (shortstr): MIME content encoding
- application_headers: (table): Message header field table: dict[str, str|int|Decimal|datetime|dict]
- 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

ack() → None
Acknowledge message

This is a convenience method which calls `self.channel.basic_ack()`

reject(*requeue*) → None
Reject message

This is a convenience method which calls `self.channel.basic_reject()`

Parameters **requeue** (*bool*) – requeue if True else discard the message

CHAPTER 4

amqpy.consumer module

CHAPTER 5

amqpy.spec module

amqpy.spec.**FRAME_MIN_SIZE = 4096**
The default, minimum frame size that both the client and server must be able to handle

class amqpy.spec.**FrameType**
Bases: object

This class contains frame-related constants

METHOD, HEADER, BODY, and HEARTBEAT are all frame type constants which make up the first byte of every frame. The END constant is the termination value which is the last byte of every frame.

class amqpy.spec.**basic_return_t**(*tuple*)
Bases: tuple

namedtuple basic_return_t(reply_code, reply_text, exchange, routing_key, message)

class amqpy.spec.**method_t**(*tuple*)
Bases: tuple

namedtuple method_t(class_id, method_id)

class amqpy.spec.**queue_declare_ok_t**(*tuple*)
Bases: tuple

namedtuple queue_declare_ok_t(queue, message_count, consumer_count)

CHAPTER 6

amqpy.proto module

High-level representations of AMQP protocol objects

class amqpy.proto.**Frame**
Bases: object

AMQP frame

A *Frame* represents the lowest-level packet of data specified by the AMQP 0.9.1 wire-level protocol. All methods and messages are packed into one or more frames before being sent to the peer.

The format of the AMQP frame is as follows:

```
offset:          0      1         3         7                  size+7      size+8
                 +------+---------+---------+------------------+-----------+
                 | type | channel |  size   |  --- payload ---  | frame-end |
                 +------+---------+---------+------------------+-----------+
size (bytes)         1       2         4             size             1
```

channel
@property

Get frame channel number

Returns channel number

Return type int

data
raw frame data; can be manually manipulated at any time

Type bytearray

frame_type
@property

Get frame type

Returns frame type

Return type int

payload
@property

Get frame payload

Returns payload

Return type bytearray

payload_size
@property

Get frame payload size

Returns payload size

Return type int

__init__ (*frame_type=None*, *channel=0*, *payload=b''*)
Create new Frame

Leave all three parameters as default to create an empty frame whose *data* can be manually written to afterwards.

Parameters

- **frame_type** (*int*) – frame type
- **channel** (*int*) – associated channel number
- **payload** (*bytes or bytearray*) – frame payload

class amqpy.proto.**Method**
Bases: object

AMQP method

The AMQP 0.9.1 protocol specifies communication as sending and receiving "methods". Methods consist of a "class-id" and "method-id" and are represented by a *method_t* namedtuple in amqpy. Methods are packed into the payload of a *FrameType.METHOD* frame, and most methods can be fully sent in a single frame. If the method specified to be carrying content (such as a message), the method frame is followed by additional frames: a *FrameType.HEADER* frame, then zero or more *FrameType.BODY* frames.

The format of the *FrameType.METHOD* frame's payload is as follows:

```
offset:          0          2           4
                 +----------+-----------+-------------- - -
                 | class-id | method-id | arguments...
                 +----------+-----------+-------------- - -
size (bytes):          2          2          variable
```

The format of the *FrameType.HEADER* frame's payload is as follows:

```
offset:          0          2        4           12                14
                 +----------+--------+-----------+-----------------+------------------- - -
                 | class-id | weight | body size | property flags | property list...
                 +----------+--------+-----------+-----------------+------------------- - -
size (bytes):          2          2          8               2            variable
```

The format of the *FrameType.BODY* frame's payload is simply raw binary data of the message body.

channel_id

Type int

complete
@property

Check if the message that is carried by this method has been completely assembled, i.e. the expected number of bytes have been loaded

This method is intended to be called when constructing a *Method* from incoming data.

Returns True if method is complete, else False

Return type bool

content

Type Message or None

method_type

Type amqpy.spec.method_t

__init__(*method_type=None*, *args=None*, *content=None*, *channel_id=None*)

Parameters

- **method_type** (method_t) – method type
- **args** (*AMQPReader or AMQPWriter or None*) – method args
- **content** (*Message or None*) – content
- **channel_id** (*int or None*) – the associated channel ID, if any

dump_body_frame(*chunk_size*) → generator[amqpy.proto.Frame]
Create a body frame

This method is intended to be called when sending frames for an already-completed *Method*.

Parameters **chunk_size** (*int*) – body chunk size in bytes; this is typically the maximum frame size - 8

Returns generator of *FrameType.BODY* frames

Return type generator[amqpy.proto.Frame]

dump_header_frame() → amqpy.proto.Frame
Create a header frame

This method is intended to be called when sending frames for an already-completed *Method*.

Returns *FrameType.HEADER* frame

Return type *amqpy.proto.Frame*

dump_method_frame() → amqpy.proto.Frame
Create a method frame

This method is intended to be called when sending frames for an already-completed *Method*.

Returns *FrameType.METHOD* frame

Return type *amqpy.proto.Frame*

load_body_frame(*frame*) → None
Add content to partial method

This method is intended to be called when constructing a *Method* from incoming data.

Parameters **frame** (amqpy.proto.Frame) – *FrameType.BODY* frame

load_header_frame(*frame*) → None
Add header to partial method

This method is intended to be called when constructing a *Method* from incoming data.

Parameters **frame** (amqpy.proto.Frame) – *FrameType.HEADER* frame

load_method_frame(*frame*) → None
Load method frame payload data

This method is intended to be called when constructing a *Method* from incoming data.

After calling, *self.method_type*, *self.args*, and *self.channel_id* will be loaded with data from the frame.

Parameters **frame** (amqpy.proto.Frame) – *FrameType.METHOD* frame

CHAPTER 7

amqpy.exceptions module

AMQP uses exceptions to handle errors:

- Any operational error (message queue not found, insufficient access rights, etc.) results in a channel exception.
- Any structural error (invalid argument, bad sequence of methods, etc.) results in a connection exception.

According to the AMQP specification, an exception closes the associated channel or connection, and returns a reply code and reply text to the client. However, amqpy will automatically re-open the channel after a channel error.

exception amqpy.exceptions.**Timeout**
Bases: TimeoutError

General AMQP operation timeout

exception amqpy.exceptions.**ContentTooLarge**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.RecoverableChannelError

The client attempted to transfer content larger than the server could accept at the present time. The client may retry at a later time.

exception amqpy.exceptions.**NoConsumers**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.RecoverableChannelError

When the exchange cannot deliver to a consumer when the immediate flag is set. As a result of pending data on the queue or the absence of any consumers of the queue.

exception amqpy.exceptions.**ConnectionForced**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.RecoverableConnectionError

An operator intervened to close the connection for some reason. The client may retry at some later date.

exception amqpy.exceptions.**InvalidPath**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableConnectionError

The client tried to work with an unknown virtual host.

exception amqpy.exceptions.**AccessRefused**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableChannelError

The client attempted to work with a server entity to which it has no access due to security settings.

exception amqpy.exceptions.**NotFound**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableChannelError

The client attempted to work with a server entity that does not exist.

exception amqpy.exceptions.**ResourceLocked**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.RecoverableChannelError

The client attempted to work with a server entity to which it has no access because another client is working with it.

exception amqpy.exceptions.**PreconditionFailed**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableChannelError

The client requested a method that was not allowed because some precondition failed.

exception amqpy.exceptions.**FrameError**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableConnectionError

The sender sent a malformed frame that the recipient could not decode. This strongly implies a programming error in the sending peer.

exception amqpy.exceptions.**FrameSyntaxError**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableConnectionError

The sender sent a frame that contained illegal values for one or more fields. This strongly implies a programming error in the sending peer.

exception amqpy.exceptions.**InvalidCommand**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableConnectionError

The client sent an invalid sequence of frames, attempting to perform an operation that was considered invalid by the server. This usually implies a programming error in the client.

exception amqpy.exceptions.**ChannelNotOpen**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableConnectionError

The client attempted to work with a channel that had not been correctly opened. This most likely indicates a fault in the client layer.

exception amqpy.exceptions.**UnexpectedFrame**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)
Bases: amqpy.exceptions.IrrecoverableConnectionError

The peer sent a frame that was not expected, usually in the context of a content header and body. This strongly indicates a fault in the peer's content processing.

exception amqpy.exceptions.**ResourceError**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)

Bases: amqpy.exceptions.RecoverableConnectionError

The server could not complete the method because it lacked sufficient resources. This may be due to the client creating too many of some type of entity.

exception amqpy.exceptions.**NotAllowed**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)

Bases: amqpy.exceptions.IrrecoverableConnectionError

The client tried to work with some entity in a manner that is prohibited by the server, due to security settings or by some other criteria.

exception amqpy.exceptions.**AMQPNotImplementedError**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)

Bases: amqpy.exceptions.IrrecoverableConnectionError

The client tried to use functionality that is not implemented in the server.

exception amqpy.exceptions.**InternalError**(*reply_text=None*, *method_type=None*, *method_name=None*, *reply_code=None*, *channel_id=None*)

Bases: amqpy.exceptions.IrrecoverableConnectionError

The server could not complete the method because of an internal error. The server may require intervention by an operator in order to resume normal operations.

CHAPTER 8

Introduction

amqpy is a pure-Python AMQP 0.9.1 client library for Python >= 3.2.0 (including PyPy3) with a focus on:

- stability and reliability
- well-tested and thoroughly documented code
- clean, correct design
- 100% compliance with the AMQP 0.9.1 protocol specification

It has very good performance, as AMQP 0.9.1 is a very efficient binary protocol, but does not sacrifice clean design and testability to save a few extra CPU cycles.

This library is actively maintained and has a zero bug policy. Please submit issues and pull requests, and bugs will be fixed immediately.

The current API is not final, but will progressively get more stable as version 1.0.0 is approached.

8.1 Guarantees

This library makes the following guarantees:

- Semantic versioning is strictly followed
- Compatible with Python >= 3.2.0 and PyPy3 >= 2.3.1 (Python 3.2.5)
- AMQP 0.9.1 compliant

CHAPTER 9

Quickstart

amqpy is easy to install, and there are no dependencies:

```
pip install amqpy
```

amqpy is easy to use:

```
from amqpy import Connection, Message, AbstractConsumer, Timeout

conn = Connection()  # connect to guest:guest@localhost:5672 by default

ch = conn.channel()

# declare an exchange and queue, and bind the queue to the exchange
ch.exchange_declare('test.exchange', 'direct')
ch.queue_declare('test.q')
ch.queue_bind('test.q', exchange='test.exchange', routing_key='test.q')

# publish a few messages, which will get routed to the queue bound to the routing key "test.q"
ch.basic_publish(Message('hello world 1'), exchange='test.exchange', routing_key='test.q')
ch.basic_publish(Message('hello world 2'), exchange='test.exchange', routing_key='test.q')
ch.basic_publish(Message('hello world 3'), exchange='test.exchange', routing_key='test.q')

# get a message from the queue
msg = ch.basic_get('test.q')

# don't forget to acknowledge it
msg.ack()
```

Let's create a consumer:

```
class Consumer(AbstractConsumer):
    def run(self, msg: Message):
        print('Received a message: {}'.format(msg.body))
        msg.ack()

consumer = Consumer(ch, 'test.q')
consumer.declare()

# wait for events, which will receive delivered messages and call any consumer callbacks
while True:
    conn.drain_events(timeout=None)
```

CHAPTER 10

Notes

Any AMQP 0.9.1-compliant server is supported, but RabbitMQ is our primary target. Apache Qpid is confirmed to work, but only with "anonymous" authentication. A CRAM-MD5 auth mechanism is currently being developed and will be released shortly.

CHAPTER 11

Features

- Draining events from multiple channels `Connection.drain_events()`
- SSL is fully supported, it is highly recommended to use SSL when connecting to servers over the Internet.
- Support for timeouts
- Support for manual and automatic heartbeats
- Fully thread-safe. Use one global connection and open one channel per thread.

Supports RabbitMQ extensions:

- Publisher confirms: enable with `Channel.confirm_select()`, then use `Channel.basic_publish_confirm`
- Exchange to exchange bindings: `Channel.exchange_bind()` and `Channel.exchange_unbind()`
- Consumer cancel notifications: by default a cancel results in `ChannelError` being raised, but not if an `on_cancel` callback is passed to `basic_consume`

CHAPTER 12

Testing

amqpy uses the excellent **tox** and **pytest** frameworks. To run all tests, simply install a local RabbitMQ server. No additional configuration is necessary for RabbitMQ. Then run in the project root:

```
$ pip install pytest
$ py.test
```

CHAPTER 13

Indices and Tables

- genindex
- modindex
- search

a

Symbols

A

B

C

D

E

F