# CoolAMQP Documentation

***Release 1.2.15***

**DMS Serwis s.c.**

**Sep 24, 2021**

# CONTENTS


**1 CoolAMQP cluster** **1**

**2 Tutorial** **5**
2.1 Publishing and consuming . . . . . 8

**3 Caveats** **13**
3.1 memoryviews . . . . . 13

**4 Glossary of all AMQP frames** **15**

**5 Usage basics** **41**

**6 Frame tracing** **45**
6.1 LoggingFrameTracer . . . . . 45
6.2 HoldingFrameTracer . . . . . 46

**7 Quick FAQ** **47**

**8 Indices and tables** **49**

**Index** **51**

CHAPTER

# ONE

# COOLAMQP CLUSTER

class coolamqp.clustering.**Cluster**(*nodes*, *on_fail=None*, *extra_properties=None*, *log_frames=None*, *name=None*, *on_blocked=None*, *tracer=None*)

Frontend for your AMQP needs.

This has ListenerThread.

Call .start() to connect to AMQP.

It is not safe to fork() after .start() is called, but it's OK before.

**Parameters**

- **nodes** – list of nodes, or a single node. For now, only one is supported.
- **on_fail** – callable/0 to call when connection fails in an unclean way. This is a one-shot
- **extra_properties** – refer to documentation in [/coolamqp/connection/connection.py] Connection.__init__
- **log_frames** (tp.Optional[*coolamqp.tracing.BaseFrameTracer*]) – an object that supports logging each and every frame CoolAMQP sends and receives from the broker
- **name** – name to appear in log items and prctl() for the listener thread
- **on_blocked** – callable to call when ConnectionBlocked/ConnectionUnblocked is received. It will be called with a value of True if connection becomes blocked, and False upon an unblock
- **tracer** – tracer, if opentracing is installed

**bind**(*queue*, *exchange*, *routing_key*, *persistent=False*, *span=None*, *dont_trace=False*)

Bind a queue to an exchange

**consume**(*queue*, *on_message=None*, *span=None*, *dont_trace=False*, **args*, ***kwargs*)

Start consuming from a queue.

args and kwargs will be passed to Consumer constructor (coolamqp.attaches.consumer.Consumer). Don't use future_to_notify - it's done here!

Take care not to lose the Consumer object - it's the only way to cancel a consumer!

**Parameters**

- **queue** – Queue object, being consumed from right now. Note that name of anonymous queue might change at any time!
- **on_message** – callable that will process incoming messages if you leave it at None, messages will be .put into self.events
- **span** – optional span, if opentracing is installed

- **dont_trace** – if True, this won't output a span

**Return type** Tuple[*Consumer*, Future]

**Returns** a tuple (Consumer instance, and a Future), that tells, when consumer is ready

**declare**(*obj*, *persistent=False*, *span=None*, *dont_trace=False*)
Declare a Queue/Exchange

**Parameters**

- **obj** (*tp.Union[Queue, Exchange]*) – Queue/Exchange object
- **persistent** (*bool*) – should it be redefined upon reconnect?
- **span** (*tp.Optional[opentracing.Span]*) – optional parent span, if opentracing is installed
- **dont_trace** (*bool*) – if True, a span won't be output

**Return type** concurrent.futures.Future

**Returns** Future

**delete_queue**(*queue*)
Delete a queue.

**Parameters** **queue** (*coolamqp.objects.Queue*) – Queue instance that represents what to delete

**Return type** Future

**Returns** a Future (will succeed with None or fail with AMQPError)

**drain**(*timeout*, *span=None*, *dont_trace=False*)
Return an Event.

**Parameters**

- **timeout** – time to wait for an event. 0 means return immediately. None means block forever
- **span** – optional parent span, if opentracing is installed
- **dont_trace** – if True, this span won't be traced

**Return type** Event

**Returns** an Event instance. NothingMuch is returned when there's nothing within a given timoeout

**publish**(*message*, *exchange=None*, *routing_key=''*, *tx=None*, *confirm=None*, *span=None*, *dont_trace=False*)
Publish a message.

**Parameters**

- **message** (Message) – Message to publish
- **exchange** (*tp.Union[Exchange, str, bytes]*) – exchange to use. Default is the "direct" empty-name exchange.
- **routing_key** (*tp.Union[str, bytes]*) – routing key to use
- **confirm** (*tp.Optional[bool]*) – Whether to publish it using confirms/transactions. If you choose so, you will receive a Future that can be used to check it broker took responsibility for this message. Note that if tx if False, and message cannot be delivered to broker at once, it will be discarded

- **tx** (*tp.Optional[bool]*) – deprecated, alias for confirm
- **span** (*tp.Optional[opentracing.Span]*) – optionally, current span, if opentracing is installed
- **dont_trace** (*bool*) – if set to True, a span won't be generated

**Return type** tp.Optional[Future]

**Returns** Future to be finished on completion or None, is confirm/tx was not chosen

**shutdown**(*wait=True*)

Terminate all connections, release resources - finish the job.

**Parameters** **wait** (bool) – block until this is done

**Raises** **RuntimeError** – if called without start() being called first

**Return type** None

**start**(*wait=True*, *timeout=10.0*)

Connect to broker. Initialize Cluster.

Only after this call is Cluster usable. It is not safe to fork after this.

**Parameters**

- **wait** (bool) – block until connection is ready
- **timeout** (float) – timeout to wait until the connection is ready. If it is not, a ConnectionDead error will be raised

**Raises**

- **RuntimeError** – called more than once
- **ConnectionDead** – failed to connect within timeout

**Return type** None

CHAPTER

TWO

# TUTORIAL

If you want to connect to an AMQP broker, you need: * its address (and port) * login and password * name of the virtual host

An idea of a heartbeat interval would be good, but you can do without. Since CoolAMQP will support clusters in the future, you should define the nodes first. You can do it using _NodeDefinition_. See NodeDefinition's documentation for alternative ways to do this, but here we will use the AMQP connection string.

**class** coolamqp.objects.**NodeDefinition**(**args*, ***kwargs*)

Definition of a reachable AMQP node.

This object is hashable.

```
>>> a = NodeDefinition(host='192.168.0.1', user='admin', password='password',
>>>                    virtual_host='vhost')
```

or

```
>>> a = NodeDefinition('192.168.0.1', 'admin', 'password')
```

or

```
>>> a = NodeDefinition('amqp://user:password@host/virtual_host')
```

or

```
>>> a = NodeDefinition('amqp://user:password@host:port/virtual_host', hearbeat=20)
```

AMQP connection string may be either bytes or str/unicode

**Additional keyword parameters that can be specified:** heartbeat - heartbeat interval in seconds port - TCP port to use. Default is 5672

**Raises** **ValueError** – invalid parameters

```
from coolamqp.objects import NodeDefinition

node = NodeDefinition('amqp://user@password:host/vhost')
```

Cluster instances are used to interface with the cluster (or a single broker). It accepts a list of nodes:

```
from coolamqp.clustering import Cluster
cluster = Cluster([node], name='My Cluster')
cluster.start(wait=True)
```

*wait=True* will block until connection is completed. After this, you can use other methods.

*name* is optional. If you specify it, and have setproctitle installed, the thread will receive a provided label, postfixed by **AMQP listener thread**.

**class** coolamqp.clustering.**Cluster**(*nodes*, *on_fail=None*, *extra_properties=None*, *log_frames=None*, *name=None*, *on_blocked=None*, *tracer=None*)

Frontend for your AMQP needs.

This has ListenerThread.

Call .start() to connect to AMQP.

It is not safe to fork() after .start() is called, but it's OK before.

**Parameters**

- **nodes** – list of nodes, or a single node. For now, only one is supported.
- **on_fail** – callable/0 to call when connection fails in an unclean way. This is a one-shot
- **extra_properties** – refer to documentation in [/coolamqp/connection/connection.py] Connection.__init__
- **log_frames** (tp.Optional[*coolamqp.tracing.BaseFrameTracer*]) – an object that supports logging each and every frame CoolAMQP sends and receives from the broker
- **name** – name to appear in log items and prctl() for the listener thread
- **on_blocked** – callable to call when ConnectionBlocked/ConnectionUnblocked is received. It will be called with a value of True if connection becomes blocked, and False upon an unblock
- **tracer** – tracer, if opentracing is installed

**bind**(*queue*, *exchange*, *routing_key*, *persistent=False*, *span=None*, *dont_trace=False*)

Bind a queue to an exchange

**consume**(*queue*, *on_message=None*, *span=None*, *dont_trace=False*, **args*, ***kwargs*)

Start consuming from a queue.

args and kwargs will be passed to Consumer constructor (coolamqp.attaches.consumer.Consumer). Don't use future_to_notify - it's done here!

Take care not to lose the Consumer object - it's the only way to cancel a consumer!

**Parameters**

- **queue** – Queue object, being consumed from right now. Note that name of anonymous queue might change at any time!
- **on_message** – callable that will process incoming messages if you leave it at None, messages will be .put into self.events
- **span** – optional span, if opentracing is installed
- **dont_trace** – if True, this won't output a span

**Return type** Tuple[*Consumer*, Future]

**Returns** a tuple (Consumer instance, and a Future), that tells, when consumer is ready

**declare**(*obj*, *persistent=False*, *span=None*, *dont_trace=False*)

Declare a Queue/Exchange

**Parameters**

- **obj** (*tp.Union[Queue, Exchange]*) – Queue/Exchange object
- **persistent** (*bool*) – should it be redefined upon reconnect?
- **span** (*tp.Optional[opentracing.Span]*) – optional parent span, if opentracing is installed
- **dont_trace** (*bool*) – if True, a span won't be output

**Return type** concurrent.futures.Future

**Returns** Future

**delete_queue**(*queue*)
Delete a queue.

**Parameters** **queue** (*coolamqp.objects.Queue*) – Queue instance that represents what to delete

**Return type** Future

**Returns** a Future (will succeed with None or fail with AMQPError)

**drain**(*timeout*, *span=None*, *dont_trace=False*)
Return an Event.

**Parameters**

- **timeout** – time to wait for an event. 0 means return immediately. None means block forever
- **span** – optional parent span, if opentracing is installed
- **dont_trace** – if True, this span won't be traced

**Return type** Event

**Returns** an Event instance. NothingMuch is returned when there's nothing within a given timoeout

**publish**(*message*, *exchange=None*, *routing_key=''*, *tx=None*, *confirm=None*, *span=None*, *dont_trace=False*)
Publish a message.

**Parameters**

- **message** (Message) – Message to publish
- **exchange** (*tp.Union[Exchange, str, bytes]*) – exchange to use. Default is the "direct" empty-name exchange.
- **routing_key** (*tp.Union[str, bytes]*) – routing key to use
- **confirm** (*tp.Optional[bool]*) – Whether to publish it using confirms/transactions. If you choose so, you will receive a Future that can be used to check it broker took responsibility for this message. Note that if tx if False, and message cannot be delivered to broker at once, it will be discarded
- **tx** (*tp.Optional[bool]*) – deprecated, alias for confirm
- **span** (*tp.Optional[opentracing.Span]*) – optionally, current span, if opentracing is installed
- **dont_trace** (*bool*) – if set to True, a span won't be generated

**Return type** tp.Optional[Future]

**Returns** Future to be finished on completion or None, is confirm/tx was not chosen

**shutdown**(*wait=True*)
Terminate all connections, release resources - finish the job.

**Parameters** **wait** (bool) – block until this is done

**Raises** **RuntimeError** – if called without start() being called first

**Return type** None

**start**(*wait=True*, *timeout=10.0*)
Connect to broker. Initialize Cluster.

Only after this call is Cluster usable. It is not safe to fork after this.

**Parameters**

- **wait** (bool) – block until connection is ready
- **timeout** (float) – timeout to wait until the connection is ready. If it is not, a ConnectionDead error will be raised

**Raises**

- **RuntimeError** – called more than once
- **ConnectionDead** – failed to connect within timeout

**Return type** None

## 2.1 Publishing and consuming

Connecting is boring. After we do, we want to do something! Let's try sending a message, and receiving it. To do that, you must first define a queue, and register a consumer.

```
from coolamqp.objects import Queue

queue = Queue(u'my_queue', auto_delete=True, exclusive=True)

consumer, consume_confirm = cluster.consume(queue, no_ack=False)
consume_confirm.result()    # wait for consuming to start
```

This will create an auto-delete and exclusive queue. After than, a consumer will be registered for this queue. _no_ack=False_ will mean that we have to manually confirm messages.

You can specify a callback, that will be called with a message if one's received by this consumer. Since we did not do that, this will go to a generic queue belonging to _Cluster_.

_consumer_ is a _Consumer_ object. This allows us to do some things with the consumer (such as setting QoS), but most importantly it allows us to cancel it later. _consume_confirm_ is a _Future_, that will succeed when AMQP _basic.consume-ok_ is received.

To send a message we need to construct it first, and later publish:

```
from coolamqp.objects import Message

msg = Message(b'hello world', properties=Message.Properties())
cluster.publish(msg, routing_key=u'my_queue')
```

**class** coolamqp.objects.**Message**(*body*, *properties=None*)
An AMQP message. Has a binary body, and some properties.

Properties is a highly regularized class - see coolamqp.framing.definitions.BasicContentPropertyList for a list of possible properties.

**Parameters**

- **body** (*anything with a buffer interface*) – stream of octets
- **properties** (*MessageProperties instance, None or a dict (SLOW!)*) – AMQP properties to be sent along. default is 'no properties at all' You can pass a dict - it will be passed to MessageProperties, but it's slow - don't do that.

**Properties**
alias of `coolamqp.framing.definitions.BasicContentPropertyList`

This creates a message with no properties, and sends it through default (direct) exchange to our queue. Note that CoolAMQP simply considers your messages to be bags of bytes + properties. It will not modify them, nor decode, and will always expect and return bytes.

To actually get our message, we need to start a consumer first. To do that, just invoke:

```
cons, fut = cluster.consume(Queue('name of the queue'), **kwargs)
```

Where kwargs are passed directly to Consumer class. **cons** is a Consumer object, and **fut** is a Future that will happen when listening has been registered on target server.

**class** coolamqp.attaches.**Consumer**(*queue*, *on_message*, *span=None*, *no_ack=True*, *qos=None*, *cancel_on_failure=False*, *future_to_notify=None*, *fail_on_first_time_resource_locked=False*, *body_receive_mode=0*)

This object represents a consumer in the system.

Consumer may reside on any AMQP broker, this is to be decided by CoolAMQP. Consumer, when created, has the state of ST_SYNCING. CoolAMQP will try to declare the consumer where it makes most sense for it to be.

If it succeeds, the consumer will enter state ST_ONLINE, and callables on_start will be called. This means that broker has confirmed that this consumer is operational and receiving messages.

Note that does not attempt to cancel consumers, or any of such nonsense. Having a channel per consumer gives you the unique possibility of simply closing the channel. Since this implies cancelling the consumer, here you go.

**WARNING: READ DEFAULT VALUES IN CONSTRUCTOR! TAKE CARE WHAT YOUR CONSUMERS DO!**

You can subscribe to be informed when the consumer is cancelled (for any reason, server or client side) with:

```
>>> con, fut = Cluster.consume(...)
>>> def im_called_on_cancel_for_any_reason():   # must have arity of 0
>>>     ..
>>> con.on_cancel.add(im_called_on_cancel_for_any_reason)
>>> con.cancel()
```

Or, if RabbitMQ is in use, you can be informed upon a Consumer Cancel Notification:

```
>>> con.on_broker_cancel.add(im_cancelled_by_broker)
```

**Parameters**

- **queue** (*coolamqp.objects.Queue*) – Queue object, being consumed from right now. Note that name of anonymous queue might change at any time!

- **on_message** (*callable(ReceivedMessage instance)*) – callable that will process incoming messages
- **span** – optional span, if opentracing is installed
- **no_ack** (*bool*) – Will this consumer require acknowledges from messages?
- **qos** (*tuple(int, int) or tuple(None, int) or int*) – a tuple of (prefetch size, prefetch window) for this consumer, or an int (prefetch window only). If an int is passed, prefetch size will be set to 0 (which means undefined), and this int will be used for prefetch window
- **cancel_on_failure** (*bool*) – Consumer will cancel itself when link goes down
- **future_to_notify** (*concurrent.futures.Future*) – Future to succeed when this consumer goes online for the first time. This future can also raise with AMQPError if it fails to.
- **fail_on_first_time_resource_locked** (*bool*) – When consumer is declared for the first time, and RESOURCE_LOCKED is encountered, it will fail the future with ResourceLocked, and consumer will cancel itself. By default it will retry until success is made. If the consumer doesn't get the chance to be declared - because of a connection fail - next reconnect will consider this to be SECOND declaration, ie. it will retry ad infinitum
- **body_receive_mode** (*a property of BodyReceiveMode*) – how should message.body be received. This has a performance impact

**cancel**()
Cancel the customer.

.ack() or .nack() for messages from this customer will have no effect.

**Return type** Future

**Returns** a Future to tell when it's done. The future will always succeed - sooner, or later. NOTE: Future is OK'd when entire channel is destroyed

**on_broker_cancel**
public, called on Customer Cancel Notification

**on_cancel**
public, called on cancel for any reason

**on_close**(*payload=None*)
Handle closing the channel. It sounds like an exception…

This is done in two steps: 1. self.state <- ST_OFFLINE, on_event(EV_OFFLINE) upon detecting

that no more messages will be there

2. self.channel_id <- None, channel is returned to Connection - c hannel has been physically torn down

Note, this can be called multiple times, and eventually with None.

**Return type** None

**on_delivery**(*sth*)
Callback for delivery-related shit

**Parameters** **sth** – AMQPMethodFrame WITH basic-deliver, AMQPHeaderFrame or AMQPBodyFrame

`on_operational`(*operational*)
[EXTEND ME] Called by internal methods (on_*) when channel has achieved (or lost) operational status.

If this is called with operational=True, then for sure it will be called with operational=False.

This will, therefore, get called an even number of times.

**Called by Channeler, when:**

- **Channeler.on_close gets called and state is ST_ONLINE** on_close registers ChannelClose, ChannelCloseOk, BasicCancel

**Parameters** `operational` (`bool`) – True if channel has just become operational, False if it has just become useless.

**Return type** `None`

`on_setup`(*payload*)
Called with different kinds of frames - during setup

**Return type** None

`set_qos`(*prefetch_size*, *prefetch_count*)
Set new QoS for this consumer.

**Parameters**

- `prefetch_size` (`int`) – prefetch in octets
- `prefetch_count` (`int`) – prefetch in whole messages

**Return type** `None`

CHAPTER

THREE

# CAVEATS

Things to look out for

## 3.1 memoryviews

Since CoolAMQP tries to be fast, it uses memoryviews everywhere. **ReceivedMessage** properties, and message properties therefore, are memoryviews. So, it you wanted to read the routing key a message was sent with, or message's encoding, you should do:

```
received_msg.routing_key.to_bytes()
received_msg.properties.content_encoding.to_bytes()
```

Only the **body** property of the message will be a byte object (and not even that it you explicitly ask otherwise).

Note that YOU, when sending messages, should not use memoryviews. Pass proper byte objects and text objects as required.

**AMQPError**'s returned to you via futures will also have memoryviews as **reply_text**, although they will properly display that once __repr__ or __str__ is called on them.

It was considered whether to unserialize short fields, such as **routing_key** or **exchange**, but it was decided against. Creating a new memoryview carries at least much overhead as an empty string, but there's no need to copy. Plus, it's not known whether you will use these strings at all!

If you need to, you got memoryviews. Plus they support the **__eq__** protocol, which should cover most use cases without even converting.

CHAPTER

FOUR

# GLOSSARY OF ALL AMQP FRAMES

class coolamqp.framing.definitions.**ConnectionBlocked**(*reason*)

This method indicates that a connection has been blocked

and does not accept new publishes.

**get_size**()

Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)

Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ConnectionClose**(*reply_code*, *reply_text*, *class_id*, *method_id*)

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.

**Parameters**

- **class_id** (*int, 16 bit unsigned (class-id in AMQP)*) – Failing method class When the close is provoked by a method exception, this is the class of the method.
- **method_id** (*int, 16 bit unsigned (method-id in AMQP)*) – Failing method id When the close is provoked by a method exception, this is the ID of the method.

**get_size**()

Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ConnectionCloseOk**
Confirm a connection close

This method confirms a Connection.Close method and tells the recipient that it is safe to release resources for the connection and close the socket.

class coolamqp.framing.definitions.**ConnectionOpen**(*virtual_host*)
Open connection to virtual host

This method opens a connection to a virtual host, which is a collection of resources, and acts to separate multiple application domains within a server. The server may apply arbitrary limits per virtual host, such as the number of each type of entity that may be used, per connection and/or in total.

**Parameters** **virtual_host** (*binary type (max length 255) (path in AMQP)*) – Virtual host name The name of the virtual host to work with.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ConnectionOpenOk**
Signal that connection is ready

This method signals to the client that the connection is ready for use.

class coolamqp.framing.definitions.**ConnectionStart**(*version_major*, *version_minor*, *server_properties*, *mechanisms*, *locales*)
Start connection negotiation

This method starts the connection negotiation process by telling the client the protocol version that the server proposes, along with a list of security mechanisms which the client can use for authentication.

**Parameters**

- **version_major** (*int, 8 bit unsigned (octet in AMQP)*) – Protocol major version The major version number can take any value from 0 to 99 as defined in the AMQP specification.

- **version_minor** (*int, 8 bit unsigned (octet in AMQP)*) – Protocol minor version The minor version number can take any value from 0 to 99 as defined in the AMQP specification.
- **server_properties** (*table. See coolamqp.uplink.framing.field_table (peer-properties in AMQP)*) – Server properties The properties SHOULD contain at least these fields: "host", specifying the server host name or address, "product", giving the name of the server product, "version", giving the name of the server version, "platform", giving the name of the operating system, "copyright", if appropriate, and "information", giving other general information.
- **mechanisms** (*binary type (longstr in AMQP)*) – Available security mechanisms A list of the security mechanisms that the server supports, delimited by spaces.
- **locales** (*binary type (longstr in AMQP)*) – Available message locales A list of the message locales that the server supports, delimited by spaces. The locale defines the language in which the server will send reply texts.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**ConnectionSecure**(*challenge*)
Security mechanism challenge

The SASL protocol works by exchanging challenges and responses until both peers have received sufficient information to authenticate each other. This method challenges the client to provide more information.

**Parameters** **challenge** (*binary type (longstr in AMQP)*) – Security challenge data Challenge information, a block of opaque binary data passed to the security mechanism.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ConnectionStartOk**(*client_properties*, *mechanism*, *response*, *locale*)

Select security mechanism and locale

This method selects a SASL security mechanism.

**Parameters**

- **client_properties** (*table. See coolamqp.uplink.framing.field_table (peer-properties in AMQP)*) – Client properties The properties SHOULD contain at least these fields: "product", giving the name of the client product, "version", giving the name of the client version, "platform", giving the name of the operating system, "copyright", if appropriate, and "information", giving other general information.
- **mechanism** (*binary type (max length 255) (shortstr in AMQP)*) – Selected security mechanism A single security mechanisms selected by the client, which must be one of those specified by the server.
- **response** (*binary type (longstr in AMQP)*) – Security response data A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.
- **locale** (*binary type (max length 255) (shortstr in AMQP)*) – Selected message locale A single message locale selected by the client, which must be one of those specified by the server.

**get_size**()

Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)

Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ConnectionSecureOk**(*response*)

Security mechanism response

This method attempts to authenticate, passing a block of SASL data for the security mechanism at the server side.

**Parameters** **response** (*binary type (longstr in AMQP)*) – Security response data A block of opaque data passed to the security mechanism. The contents of this data are defined by the SASL security mechanism.

**get_size**()

Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**ConnectionTune**(*channel_max*, *frame_max*, *heartbeat*)
Propose connection tuning parameters

This method proposes a set of connection configuration values to the client. The client can accept and/or adjust these.

**Parameters**

- **channel_max** (*int, 16 bit unsigned (short in AMQP)*) – Proposed maximum channels Specifies highest channel number that the server permits. Usable channel numbers are in the range 1..channel-max. Zero indicates no specified limit.
- **frame_max** (*int, 32 bit unsigned (long in AMQP)*) – Proposed maximum frame size The largest frame size that the server proposes for the connection, including frame header and end-byte. The client can negotiate a lower value. Zero means that the server does not impose any specific limit but may reject very large frames if it cannot allocate resources for them.
- **heartbeat** (*int, 16 bit unsigned (short in AMQP)*) – Desired heartbeat delay The delay, in seconds, of the connection heartbeat that the server wants. Zero means the server does not want a heartbeat.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**ConnectionTuneOk**(*channel_max*, *frame_max*, *heartbeat*)
Negotiate connection tuning parameters

This method sends the client's connection tuning parameters to the server. Certain fields are negotiated, others provide capability information.

**Parameters**

- **channel_max** (*int, 16 bit unsigned (short in AMQP)*) – Negotiated maximum channels The maximum total number of channels that the client will use per connection.
- **frame_max** (*int, 32 bit unsigned (long in AMQP)*) – Negotiated maximum frame size The largest frame size that the client and server will use for the connection. Zero means that the client does not impose any specific limit but may reject very large frames if it cannot allocate resources for them. Note that the frame-max limit applies principally to content frames, where large contents can be broken into frames of arbitrary size.
- **heartbeat** (*int, 16 bit unsigned (short in AMQP)*) – Desired heartbeat delay The delay, in seconds, of the connection heartbeat that the client wants. Zero means the client does not want a heartbeat.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**ConnectionUnblocked**
This method indicates that a connection has been unblocked

and now accepts publishes.

**class** coolamqp.framing.definitions.**ChannelClose**(*reply_code*, *reply_text*, *class_id*, *method_id*)
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**

- **class_id** (*int, 16 bit unsigned (class-id in AMQP)*) – Failing method class When the close is provoked by a method exception, this is the class of the method.
- **method_id** (*int, 16 bit unsigned (method-id in AMQP)*) – Failing method id When the close is provoked by a method exception, this is the ID of the method.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**ChannelCloseOk**
Confirm a channel close

This method confirms a Channel.Close method and tells the recipient that it is safe to release resources for the channel.

**class** coolamqp.framing.definitions.**ChannelFlow**(*active*)
Enable/disable flow from peer

This method asks the peer to pause or restart the flow of content data sent by a consumer. 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. It does not affect contents returned by Basic.Get-Ok methods.

**Parameters** **active** (*bool (bit in AMQP)*) – Start/stop content frames If 1, the peer starts sending content frames. If 0, the peer stops sending content frames.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**ChannelFlowOk**(*active*)
Confirm a flow method

Confirms to the peer that a flow command was received and processed.

**Parameters** **active** (*bool (bit in AMQP)*) – Current flow setting Confirms the setting of the processed flow method: 1 means the peer will start sending or continue to send content frames; 0 means it will not.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ChannelOpen**
Open a channel for use

This method opens a channel to the server.

class coolamqp.framing.definitions.**ChannelOpenOk**
Signal that the channel is ready

This method signals to the client that the channel is ready for use.

class coolamqp.framing.definitions.**ExchangeBind**(*destination*, *source*, *routing_key*, *no_wait*, *arguments*)
Bind exchange to an exchange

This method binds an exchange to an exchange.

**Parameters**

- **destination** (*binary type (max length 255) (exchange-name in AMQP)*) – Name of the destination exchange to bind to Specifies the name of the destination exchange to bind.
- **source** (*binary type (max length 255) (exchange-name in AMQP)*) – Name of the source exchange to bind to Specifies the name of the source exchange to bind.
- **routing_key** (*binary type (max length 255) (shortstr in AMQP)*) – 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.
- **arguments** (*table. See coolamqp.uplink.framing.field_table (table in AMQP)*) – Arguments for binding A set of arguments for the binding. The syntax and semantics of these arguments depends on the exchange class.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ExchangeBindOk**
Confirm bind successful

This method confirms that the bind was successful.

class coolamqp.framing.definitions.**ExchangeDeclare**(*exchange*, *type_*, *passive*, *durable*, *auto_delete*, *internal*, *no_wait*, *arguments*)
Verify exchange exists, 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.

**Parameters**

- **exchange** (*binary type (max length 255) (exchange-name in AMQP)*) – Exchange names starting with "amq." are reserved for pre-declared and standardised exchanges. The client MAY declare an exchange starting with "amq." if the passive option is set, or the exchange already exists.
- **type** (*binary type (max length 255) (shortstr in AMQP)*) – 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.
- **passive** (*bool (bit in AMQP)*) – Do not create exchange If set, the server will reply with Declare-Ok if the exchange already exists with the same name, and raise an error if not. The client can use this to check whether an exchange exists without modifying the server state. When set, all other method fields except name and no-wait are ignored. A declare with both passive and no-wait has no effect. Arguments are compared for semantic equivalence.
- **durable** (*bool (bit in AMQP)*) – 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.
- **auto_delete** (*bool (bit in AMQP)*) – Auto-delete when unused If set, the exchange is deleted when all queues have finished using it.
- **internal** (*bool (bit in AMQP)*) – Create internal exchange If set, the exchange may not be used directly by publishers, but only when bound to other exchanges. Internal exchanges are used to construct wiring that is not visible to applications.
- **arguments** (*table. See coolamqp.uplink.framing.field_table (table in AMQP)*) – Arguments for declaration A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation.

get_size()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

write_arguments(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ExchangeDelete**(*exchange*, *if_unused*, *no_wait*)
Delete an exchange

This method deletes an exchange. When an exchange is deleted all queue bindings on the exchange are cancelled.

**Parameters**

- **exchange** (*binary type (max length 255) (exchange-name in AMQP)*) – The client must not attempt to delete an exchange that does not exist.
- **if_unused** (*bool (bit in AMQP)*) – 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.

get_size()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

write_arguments(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ExchangeDeclareOk**
Confirm exchange declaration

This method confirms a Declare method and confirms the name of the exchange, essential for automatically-named exchanges.

class coolamqp.framing.definitions.**ExchangeDeleteOk**
Confirm deletion of an exchange

This method confirms the deletion of an exchange.

class coolamqp.framing.definitions.**ExchangeUnbind**(*destination*, *source*, *routing_key*, *no_wait*, *arguments*)
Unbind an exchange from an exchange

This method unbinds an exchange from an exchange.

**Parameters**

- **destination** (*binary type (max length 255) (exchange-name in AMQP)*) – Specifies the name of the destination exchange to unbind.
- **source** (*binary type (max length 255) (exchange-name in AMQP)*) – Specifies the name of the source exchange to unbind.
- **routing_key** (*binary type (max length 255) (shortstr in AMQP)*) – Routing key of binding Specifies the routing key of the binding to unbind.

- **arguments** (*table. See coolamqp.uplink.framing.field_table (table in AMQP)*) – Arguments of binding Specifies the arguments of the binding to unbind.

get_size()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

write_arguments(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**ExchangeUnbindOk**
Confirm unbind successful

This method confirms that the unbind was successful.

class coolamqp.framing.definitions.**QueueBind**(*queue*, *exchange*, *routing_key*, *no_wait*, *arguments*)
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 direct exchange and subscription queues are bound to a topic exchange.

**Parameters**

- **queue** (*binary type (max length 255) (queue-name in AMQP)*) – Specifies the name of the queue to bind.
- **exchange** (*binary type (max length 255) (exchange-name in AMQP)*) – Name of the exchange to bind to A client MUST NOT be allowed to bind a queue to a non-existent exchange.
- **routing_key** (*binary type (max length 255) (shortstr in AMQP)*) – 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 queue name is empty, the server uses the last queue declared on the channel. If the routing key is also empty, the server uses this queue name for the routing key as well. If the queue name is provided but the routing key is empty, the server does the binding with that empty routing key. The meaning of empty routing keys depends on the exchange implementation.
- **arguments** (*table. See coolamqp.uplink.framing.field_table (table in AMQP)*) – Arguments for binding A set of arguments for the binding. The syntax and semantics of these arguments depends on the exchange class.

get_size()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**QueueBindOk**
Confirm bind successful

This method confirms that the bind was successful.

**class** coolamqp.framing.definitions.**QueueDeclare**(*queue*, *passive*, *durable*, *exclusive*, *auto_delete*, *no_wait*, *arguments*)
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.

**Parameters**

- **queue** (*binary type (max length 255) (queue-name in AMQP)*) – 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.
- **passive** (*bool (bit in AMQP)*) – Do not create queue If set, the server will reply with Declare-Ok if the queue already exists with the same name, and raise an error if not. The client can use this to check whether a queue exists without modifying the server state. When set, all other method fields except name and no-wait are ignored. A declare with both passive and no-wait has no effect. Arguments are compared for semantic equivalence.
- **durable** (*bool (bit in AMQP)*) – 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.
- **exclusive** (*bool (bit in AMQP)*) – Request an exclusive queue Exclusive queues may only be accessed by the current connection, and are deleted when that connection closes. Passive declaration of an exclusive queue by other connections are not allowed.
- **auto_delete** (*bool (bit in AMQP)*) – Auto-delete queue when unused If set, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted. Applications can explicitly delete auto-delete queues using the Delete method as normal.
- **arguments** (*table. See coolamqp.uplink.framing.field_table (table in AMQP)*) – Arguments for declaration A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**QueueDelete**(*queue*, *if_unused*, *if_empty*, *no_wait*)
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** (*binary type (max length 255) (queue-name in AMQP)*) – Specifies the name of the queue to delete.
- **if_unused** (*bool (bit in AMQP)*) – 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.
- **if_empty** (*bool (bit in AMQP)*) – Delete only if empty If set, the server will only delete the queue if it has no messages.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**QueueDeclareOk**(*queue*, *message_count*, *consumer_count*)
Confirms a queue definition

This method confirms a Declare method and confirms the name of the queue, essential for automatically-named queues.

**Parameters**

- **queue** (*binary type (max length 255) (queue-name in AMQP)*) – Reports the name of the queue. if the server generated a queue name, this field contains that name.

- **consumer_count** (*int, 32 bit unsigned (long in AMQP)*) – Number of consumers Reports the number of active consumers for the queue. Note that consumers can suspend activity (Channel.Flow) in which case they do not appear in this count.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**QueueDeleteOk**(*message_count*)
Confirm deletion of a queue

This method confirms the deletion of a queue.

**Parameters** **message_count** (*int, 32 bit unsigned (message-count in AMQP)*) – Reports the number of messages deleted.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**QueuePurge**(*queue*, *no_wait*)
Purge a queue

This method removes all messages from a queue which are not awaiting acknowledgment.

**Parameters** **queue** (*binary type (max length 255) (queue-name in AMQP)*) – Specifies the name of the queue to purge.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**QueuePurgeOk**(*message_count*)
Confirms a queue purge

This method confirms the purge of a queue.

**Parameters** **message_count** (*int, 32 bit unsigned (message-count in AMQP)*) – Reports the number of messages purged.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**QueueUnbind**(*queue*, *exchange*, *routing_key*, *arguments*)
Unbind a queue from an exchange

This method unbinds a queue from an exchange.

**Parameters**

- **queue** (*binary type (max length 255) (queue-name in AMQP)*) – Specifies the name of the queue to unbind.
- **exchange** (*binary type (max length 255) (exchange-name in AMQP)*) – The name of the exchange to unbind from.
- **routing_key** (*binary type (max length 255) (shortstr in AMQP)*) – Routing key of binding Specifies the routing key of the binding to unbind.
- **arguments** (*table. See coolamqp.uplink.framing.field_table (table in AMQP)*) – Arguments of binding Specifies the arguments of the binding to unbind.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**QueueUnbindOk**
Confirm unbind successful

This method confirms that the unbind was successful.

class coolamqp.framing.definitions.**BasicAck**(*delivery_tag*, *multiple*)
Acknowledge one or more messages

When sent by the client, this method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. When sent by server, this method acknowledges one or more messages published with the Publish method on a channel in confirm mode. The acknowledgement can be for a single message or a set of messages up to and including a specific message.

**Parameters** **multiple** (*bool (bit in AMQP)*) – Acknowledge multiple messages If set to 1, the delivery tag is treated as "up to and including", so that multiple messages can be acknowledged with a single method. If set to zero, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates acknowledgement of all outstanding messages.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicConsume**(*queue*, *consumer_tag*, *no_local*, *no_ack*, *exclusive*, *no_wait*, *arguments*)
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 declared on, or until the client cancels them.

**Parameters**

- **queue** (*binary type (max length 255) (queue-name in AMQP)*) – Specifies the name of the queue to consume from.
- **consumer_tag** (*binary type (max length 255) (consumer-tag in AMQP)*) – Specifies the identifier for the consumer. the consumer tag is local to a channel, so two clients can use the same consumer tags. If this field is empty the server will generate a unique tag.
- **exclusive** (*bool (bit in AMQP)*) – Request exclusive access Request exclusive consumer access, meaning only this consumer can access the queue.
- **arguments** (*table. See coolamqp.uplink.framing.field_table (table in AMQP)*) – Arguments for declaration A set of arguments for the consume. The syntax and semantics of these arguments depends on the server implementation.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicCancel**(*consumer_tag*, *no_wait*)
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. It may also be sent from the server to the client in the event of the consumer being unexpectedly cancelled (i.e. cancelled for any reason other than the server receiving the corresponding basic.cancel from the client). This allows clients to be notified of the loss of consumers due to events such as queue deletion. Note that as it is not a MUST for clients to accept this method from the server, it is advisable for the broker to be able to identify those clients that are capable of accepting the method, through some means of capability negotiation.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicConsumeOk**(*consumer_tag*)
Confirm a new consumer

The server provides the client with a consumer tag, which is used by the client for methods called on the consumer at a later stage.

**Parameters** **consumer_tag** (*binary type (max length 255) (consumer-tag in AMQP)*) – Holds the consumer tag specified by the client or provided by the server.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicCancelOk**(*consumer_tag*)
Confirm a cancelled consumer

This method confirms that the cancellation was completed.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicDeliver**(*consumer_tag*, *delivery_tag*, *redelivered*, *exchange*, *routing_key*)
Notify the client of a consumer message

This method delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume method, then the server responds with Deliver methods as and when messages arrive for that consumer.

**Parameters**

- **exchange** (*binary type (max length 255) (exchange-name in AMQP)*) – Specifies the name of the exchange that the message was originally published to. May be empty, indicating the default exchange.
- **routing_key** (*binary type (max length 255) (shortstr in AMQP)*) – Message routing key Specifies the routing key name specified when the message was published.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicGet**(*queue*, *no_ack*)
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** (*binary type (max length 255) (queue-name in AMQP)*) – Specifies the name of the queue to get a message from.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicGetOk**(*delivery_tag*, *redelivered*, *exchange*, *routing_key*, *message_count*)

Provide client with a message

This method delivers a message to the client following a get method. A message delivered by 'get-ok' must be acknowledged unless the no-ack option was set in the get method.

**Parameters**

- **exchange** (*binary type (max length 255) (exchange-name in AMQP)*) – Specifies the name of the exchange that the message was originally published to. If empty, the message was published to the default exchange.
- **routing_key** (*binary type (max length 255) (shortstr in AMQP)*) – Message routing key Specifies the routing key name specified when the message was published.

**get_size**()

Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)

Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicGetEmpty**

Indicate no messages available

This method tells the client that the queue has no messages available for the client.

class coolamqp.framing.definitions.**BasicNack**(*delivery_tag*, *multiple*, *requeue*)

Reject one or more incoming messages

This method allows a client to reject one or more incoming messages. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue. This method is also used by the server to inform publishers on channels in confirm mode of unhandled messages. If a publisher receives this method, it probably needs to republish the offending messages.

**Parameters**

- **multiple** (*bool (bit in AMQP)*) – Reject multiple messages If set to 1, the delivery tag is treated as "up to and including", so that multiple messages can be rejected with a single method. If set to zero, the delivery tag refers to a single message. If the multiple field is 1, and the delivery tag is zero, this indicates rejection of all outstanding messages.
- **requeue** (*bool (bit in AMQP)*) – Requeue the message If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered. Clients receiving the Nack methods should ignore this flag.

**get_size**()

Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**BasicPublish**(*exchange*, *routing_key*, *mandatory*, *immediate*)
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** (*binary type (max length 255) (exchange-name in AMQP)*) – 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.
- **routing_key** (*binary type (max length 255) (shortstr in AMQP)*) – Message routing key Specifies the routing key for the message. The routing key is used for routing messages depending on the exchange configuration.
- **mandatory** (*bool (bit in AMQP)*) – Indicate mandatory routing This flag tells the server how to react if the message cannot be routed to a queue. If this flag is set, the server will return an unroutable message with a Return method. If this flag is zero, the server silently drops the message.
- **immediate** (*bool (bit in AMQP)*) – 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.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicQos**(*prefetch_size*, *prefetch_count*, *global_*)
Specify quality of service

This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The particular properties and semantics of a qos method always depend on the content class semantics. Though the qos method could in principle apply to both peers, it is currently meaningful only for the server.

**Parameters**

- **prefetch_size** (*int, 32 bit unsigned (long in AMQP)*) – 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.
- **prefetch_count** (*int, 16 bit unsigned (short in AMQP)*) – 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.
- **global** (*bool (bit in AMQP)*) – Apply to entire connection RabbitMQ has reinterpreted this field. The original specification said: "By default the QoS settings apply to the current channel only. If this field is set, they are applied to the entire connection." Instead, RabbitMQ takes global=false to mean that the QoS settings should apply per-consumer (for new consumers on the channel; existing ones being unaffected) and global=true to mean that the QoS settings should apply per-channel.

get_size()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

write_arguments(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicQosOk**
Confirm the requested qos

This method tells the client that the requested QoS levels could be handled by the server. The requested QoS applies to all active consumers until a new QoS is defined.

class coolamqp.framing.definitions.**BasicReturn**(*reply_code*, *reply_text*, *exchange*, *routing_key*)
Return a failed message

This method returns an undeliverable message that was published with the "immediate" flag set, or an unroutable message published with the "mandatory" flag set. The reply code and text provide information about the reason that the message was undeliverable.

**Parameters**

- **exchange** (*binary type (max length 255) (exchange-name in AMQP)*) – Specifies the name of the exchange that the message was originally published to. May be empty, meaning the default exchange.
- **routing_key** (*binary type (max length 255) (shortstr in AMQP)*) – Message routing key Specifies the routing key name specified when the message was published.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicReject**(*delivery_tag*, *requeue*)
Reject an incoming message

This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

**Parameters** **requeue** (*bool (bit in AMQP)*) – Requeue the message If requeue is true, the server will attempt to requeue the message. If requeue is false or the requeue attempt fails the messages are discarded or dead-lettered.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicRecoverAsync**(*requeue*)
Redeliver unacknowledged messages

This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method is deprecated in favour of the synchronous Recover/Recover-Ok.

**Parameters** **requeue** (*bool (bit in AMQP)*) – Requeue the message If this field is zero, the message will be redelivered to the original recipient. If this bit is 1, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicRecover**(*requeue*)
Redeliver unacknowledged messages

This method asks the server to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method replaces the asynchronous Recover.

**Parameters** **requeue** (*bool (bit in AMQP)*) – Requeue the message If this field is zero, the message will be redelivered to the original recipient. If this bit is 1, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

class coolamqp.framing.definitions.**BasicRecoverOk**
Confirm recovery

This method acknowledges a Basic.Recover method.

**class** coolamqp.framing.definitions.**TxCommit**
Commit the current transaction

This method commits all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a commit.

**class** coolamqp.framing.definitions.**TxCommitOk**
Confirm a successful commit

This method confirms to the client that the commit succeeded. Note that if a commit fails, the server raises a channel exception.

**class** coolamqp.framing.definitions.**TxRollback**
Abandon the current transaction

This method abandons all message publications and acknowledgments performed in the current transaction. A new transaction starts immediately after a rollback. Note that unacked messages will not be automatically redelivered by rollback; if that is required an explicit recover call should be issued.

**class** coolamqp.framing.definitions.**TxRollbackOk**
Confirm successful rollback

This method confirms to the client that the rollback succeeded. Note that if an rollback fails, the server raises a channel exception.

**class** coolamqp.framing.definitions.**TxSelect**
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.

**class** coolamqp.framing.definitions.**TxSelectOk**
Confirm transaction mode

This method confirms to the client that the channel was successfully set to use standard transactions.

**class** coolamqp.framing.definitions.**ConfirmSelect**(*nowait*)
This method sets the channel to use publisher acknowledgements.

The client can only use this method on a non-transactional channel.

**Parameters** **nowait** (*bool (bit in AMQP)*) – 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.

**get_size**()
Calculate the size of this frame.

Needs to be overloaded, unless you're a class with IS_CONTENT_STATIC

**Return type** int

**Returns** int, size of argument section

**Raises** **RuntimeError** – this class isn't IS_CONTENT_STATIC and this method was called directly. In this case, you should have rather subclassed it.

**write_arguments**(*buf*)
Write the argument portion of this frame into target buffer.

**Parameters** **buf** (*tp.BinaryIO*) – buffer to write to

**Raises** **ValueError** – some field here is invalid!

**Return type** None

**class** coolamqp.framing.definitions.**ConfirmSelectOk**

This method confirms to the client that the channel was

successfully set to use publisher acknowledgements.

CHAPTER

# FIVE

# USAGE BASICS

First off, you need a Cluster object:

**class** coolamqp.clustering.**Cluster**(*nodes*, *on_fail=None*, *extra_properties=None*, *log_frames=None*, *name=None*, *on_blocked=None*, *tracer=None*)

Frontend for your AMQP needs.

This has ListenerThread.

Call .start() to connect to AMQP.

It is not safe to fork() after .start() is called, but it's OK before.

**Parameters**

- **nodes** – list of nodes, or a single node. For now, only one is supported.
- **on_fail** – callable/0 to call when connection fails in an unclean way. This is a one-shot
- **extra_properties** – refer to documentation in [/coolamqp/connection/connection.py] Connection.__init__
- **log_frames** (tp.Optional[*coolamqp.tracing.BaseFrameTracer*]) – an object that supports logging each and every frame CoolAMQP sends and receives from the broker
- **name** – name to appear in log items and prctl() for the listener thread
- **on_blocked** – callable to call when ConnectionBlocked/ConnectionUnblocked is received. It will be called with a value of True if connection becomes blocked, and False upon an unblock
- **tracer** – tracer, if opentracing is installed

**bind**(*queue*, *exchange*, *routing_key*, *persistent=False*, *span=None*, *dont_trace=False*)

Bind a queue to an exchange

**consume**()

Start consuming from a queue.

args and kwargs will be passed to Consumer constructor (coolamqp.attaches.consumer.Consumer). Don't use future_to_notify - it's done here!

Take care not to lose the Consumer object - it's the only way to cancel a consumer!

**Parameters**

- **queue** – Queue object, being consumed from right now. Note that name of anonymous queue might change at any time!
- **on_message** – callable that will process incoming messages if you leave it at None, messages will be .put into self.events

- **span** – optional span, if opentracing is installed
- **dont_trace** – if True, this won't output a span

**Return type** Tuple[*Consumer*, Future]

**Returns** a tuple (Consumer instance, and a Future), that tells, when consumer is ready

**declare**(*obj*, *persistent=False*, *span=None*, *dont_trace=False*)
Declare a Queue/Exchange

**Parameters**

- **obj** (*tp.Union[Queue, Exchange]*) – Queue/Exchange object
- **persistent** (*bool*) – should it be redefined upon reconnect?
- **span** (*tp.Optional[opentracing.Span]*) – optional parent span, if opentracing is installed
- **dont_trace** (*bool*) – if True, a span won't be output

**Return type** concurrent.futures.Future

**Returns** Future

**delete_queue**(*queue*)
Delete a queue.

**Parameters** **queue** (*coolamqp.objects.Queue*) – Queue instance that represents what to delete

**Return type** Future

**Returns** a Future (will succeed with None or fail with AMQPError)

**drain**()
Return an Event.

**Parameters**

- **timeout** – time to wait for an event. 0 means return immediately. None means block forever
- **span** – optional parent span, if opentracing is installed
- **dont_trace** – if True, this span won't be traced

**Return type** Event

**Returns** an Event instance. NothingMuch is returned when there's nothing within a given timoeout

**publish**(*message*, *exchange=None*, *routing_key=''*, *tx=None*, *confirm=None*, *span=None*, *dont_trace=False*)
Publish a message.

**Parameters**

- **message** (Message) – Message to publish
- **exchange** (*tp.Union[Exchange, str, bytes]*) – exchange to use. Default is the "direct" empty-name exchange.
- **routing_key** (*tp.Union[str, bytes]*) – routing key to use

- **confirm** (*tp.Optional[bool]*) – Whether to publish it using confirms/transactions. If you choose so, you will receive a Future that can be used to check it broker took responsibility for this message. Note that if tx if False, and message cannot be delivered to broker at once, it will be discarded
- **tx** (*tp.Optional[bool]*) – deprecated, alias for confirm
- **span** (*tp.Optional[opentracing.Span]*) – optionally, current span, if opentracing is installed
- **dont_trace** (*bool*) – if set to True, a span won't be generated

**Return type** tp.Optional[Future]

**Returns** Future to be finished on completion or None, is confirm/tx was not chosen

**shutdown**(*wait=True*)

Terminate all connections, release resources - finish the job.

**Parameters** **wait** (bool) – block until this is done

**Raises** **RuntimeError** – if called without start() being called first

**Return type** None

**start**(*wait=True*, *timeout=10.0*)

Connect to broker. Initialize Cluster.

Only after this call is Cluster usable. It is not safe to fork after this.

**Parameters**

- **wait** (bool) – block until connection is ready
- **timeout** (float) – timeout to wait until the connection is ready. If it is not, a ConnectionDead error will be raised

**Raises**

- **RuntimeError** – called more than once
- **ConnectionDead** – failed to connect within timeout

**Return type** None

You will need to initialize it with NodeDefinitions:

**class** coolamqp.objects.**NodeDefinition**(**args*, ***kwargs*)

Definition of a reachable AMQP node.

This object is hashable.

```
>>> a = NodeDefinition(host='192.168.0.1', user='admin', password='password',
>>>                    virtual_host='vhost')
```

or

```
>>> a = NodeDefinition('192.168.0.1', 'admin', 'password')
```

or

```
>>> a = NodeDefinition('amqp://user:password@host/virtual_host')
```

or

```
>>> a = NodeDefinition('amqp://user:password@host:port/virtual_host', hearbeat=20)
```

AMQP connection string may be either bytes or str/unicode

**Additional keyword parameters that can be specified:** heartbeat - heartbeat interval in seconds port - TCP port to use. Default is 5672

**Raises** `ValueError` – invalid parameters

You can send messages:

`class coolamqp.objects.Message(body, properties=None)`

An AMQP message. Has a binary body, and some properties.

Properties is a highly regularized class - see coolamqp.framing.definitions.BasicContentPropertyList for a list of possible properties.

**Parameters**

- **body** (*anything with a buffer interface*) – stream of octets
- **properties** (*MessageProperties instance, None or a dict (SLOW!)*) – AMQP properties to be sent along. default is 'no properties at all' You can pass a dict - it will be passed to MessageProperties, but it's slow - don't do that.

and receive them

`class coolamqp.objects.ReceivedMessage(body, exchange_name, routing_key, properties=None, delivery_tag=None, ack=None, nack=None)`

A message that was received from the AMQP broker.

It additionally has an exchange name, routing key used, it's delivery tag, and methods for ack() or nack().

Note that if the consumer that generated this message was no_ack, .ack() and .nack() are no-ops.

`ack()`

Acknowledge reception of this message.

This is a no-op if a Consumer was called with no_ack=True.

If called after an ack() or nack() was called, this will be a no-op.

`nack()`

Negatively acknowledge reception of this message.

This is a no-op if a Consumer was called with no_ack=True. If no_ack was False, the message will be requeued and redelivered by the broker

If called after an ack() or nack() was called, this will be a no-op.

CHAPTER

SIX

# FRAME TRACING

CoolAMQP allows you to trace every sent or received frame. Just provide an instance of

**class** coolamqp.tracing.**BaseFrameTracer**
An abstract do-nothing frame tracer

**on_frame**(*timestamp*, *frame*, *direction*)
Called by AMQP upon receiving a frame information

**Parameters**

- **timestamp** (*float*) – timestamp
- **frame** (coolamqp.framing.base.AMQPFrame) – frame that is sent or received
- **direction** (*str*) – either 'to_client' or 'to_server'

## 6.1 LoggingFrameTracer

To show each frame that is sent or received to the server use the following:

```
import logging

logger = logging.getLogger(__name__)

from coolamqp.tracing import LoggingFrameTracer

frame_tracer = LoggingFrameTracer(logger, logging.WARNING)

cluster = Cluster([NODE], log_frames=frame_logger)
cluster.start()
```

Documentation of the class:

**class** coolamqp.tracing.**LoggingFrameTracer**(*logger*, *log_level=30*)
A frame tracer that outputs each frame to log

**Parameters**

- **logger** – the logger to log onto
- **log_level** – the level of logging to log with

**on_frame**(*timestamp*, *frame*, *direction*)
Called by AMQP upon receiving a frame information

**Parameters**

- **timestamp** (*float*) – timestamp
- **frame** (coolamqp.framing.base.AMQPFrame) – frame that is sent or received
- **direction** (*str*) – either 'to_client' or 'to_server'

## 6.2 HoldingFrameTracer

**class** coolamqp.tracing.**HoldingFrameTracer**

A frame tracer that holds the frames in memory

**Variables** **frames** – a list of tuple (direction:str (either 'to_client' or 'to_server'), timestamp::float, frame:: AMQPFrame)

**clear**()

Clear internal frame list

**on_frame**(*timestamp*, *frame*, *direction*)

Called by AMQP upon receiving a frame information

**Parameters**

- **timestamp** (*float*) – timestamp
- **frame** (coolamqp.framing.base.AMQPFrame) – frame that is sent or received
- **direction** (*str*) – either 'to_client' or 'to_server'

CHAPTER

SEVEN

# QUICK FAQ

Q: **I'm running uWSGI and I can't publish messages. What's wrong?**

**A: Since CoolAMQP spins a thread in the background, make sure to run** uwsgi with `--enable-threads`

CHAPTER

EIGHT

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX


## A

ack() (*coolamqp.objects.ReceivedMessage method*), 44

## B

BaseFrameTracer (*class in coolamqp.tracing*), 45
BasicAck (*class in coolamqp.framing.definitions*), 30
BasicCancel (*class in coolamqp.framing.definitions*), 31
BasicCancelOk (*class in coolamqp.framing.definitions*), 32
BasicConsume (*class in coolamqp.framing.definitions*), 30
BasicConsumeOk (*class in coolamqp.framing.definitions*), 32
BasicDeliver (*class in coolamqp.framing.definitions*), 32
BasicGet (*class in coolamqp.framing.definitions*), 33
BasicGetEmpty (*class in coolamqp.framing.definitions*), 34
BasicGetOk (*class in coolamqp.framing.definitions*), 33
BasicNack (*class in coolamqp.framing.definitions*), 34
BasicPublish (*class in coolamqp.framing.definitions*), 35
BasicQos (*class in coolamqp.framing.definitions*), 35
BasicQosOk (*class in coolamqp.framing.definitions*), 36
BasicRecover (*class in coolamqp.framing.definitions*), 38
BasicRecoverAsync (*class in coolamqp.framing.definitions*), 37
BasicRecoverOk (*class in coolamqp.framing.definitions*), 38
BasicReject (*class in coolamqp.framing.definitions*), 37
BasicReturn (*class in coolamqp.framing.definitions*), 36
bind() (*coolamqp.clustering.Cluster method*), 1, 6, 41

## C

cancel() (*coolamqp.attaches.Consumer method*), 10
ChannelClose (*class in coolamqp.framing.definitions*), 20
ChannelCloseOk (*class in coolamqp.framing.definitions*), 21
ChannelFlow (*class in coolamqp.framing.definitions*), 21
ChannelFlowOk (*class in coolamqp.framing.definitions*), 21
ChannelOpen (*class in coolamqp.framing.definitions*), 22
ChannelOpenOk (*class in coolamqp.framing.definitions*), 22
clear() (*coolamqp.tracing.HoldingFrameTracer method*), 46
Cluster (*class in coolamqp.clustering*), 1, 6, 41
ConfirmSelect (*class in coolamqp.framing.definitions*), 39
ConfirmSelectOk (*class in coolamqp.framing.definitions*), 39
ConnectionBlocked (*class in coolamqp.framing.definitions*), 15
ConnectionClose (*class in coolamqp.framing.definitions*), 15
ConnectionCloseOk (*class in coolamqp.framing.definitions*), 16
ConnectionOpen (*class in coolamqp.framing.definitions*), 16
ConnectionOpenOk (*class in coolamqp.framing.definitions*), 16
ConnectionSecure (*class in coolamqp.framing.definitions*), 17
ConnectionSecureOk (*class in coolamqp.framing.definitions*), 18
ConnectionStart (*class in coolamqp.framing.definitions*), 16
ConnectionStartOk (*class in coolamqp.framing.definitions*), 18
ConnectionTune (*class in coolamqp.framing.definitions*), 19
ConnectionTuneOk (*class in coolamqp.framing.definitions*), 19
ConnectionUnblocked (*class in coolamqp.framing.definitions*), 20
consume() (*coolamqp.clustering.Cluster method*), 1, 6, 41
Consumer (*class in coolamqp.attaches*), 9

## D

declare() (*coolamqp.clustering.Cluster method*), 2, 6,

42
delete_queue() (coolamqp.clustering.Cluster method), 2, 7, 42
drain() (coolamqp.clustering.Cluster method), 2, 7, 42

## E

ExchangeBind (class in coolamqp.framing.definitions), 22
ExchangeBindOk (class in coolamqp.framing.definitions), 22
ExchangeDeclare (class in coolamqp.framing.definitions), 23
ExchangeDeclareOk (class in coolamqp.framing.definitions), 24
ExchangeDelete (class in coolamqp.framing.definitions), 24
ExchangeDeleteOk (class in coolamqp.framing.definitions), 24
ExchangeUnbind (class in coolamqp.framing.definitions), 24
ExchangeUnbindOk (class in coolamqp.framing.definitions), 25

## G

get_size() (coolamqp.framing.definitions.BasicAck method), 30
get_size() (coolamqp.framing.definitions.BasicCancel method), 31
get_size() (coolamqp.framing.definitions.BasicCancelOk method), 32
get_size() (coolamqp.framing.definitions.BasicConsume method), 31
get_size() (coolamqp.framing.definitions.BasicConsumeOk method), 32
get_size() (coolamqp.framing.definitions.BasicDeliver method), 33
get_size() (coolamqp.framing.definitions.BasicGet method), 33
get_size() (coolamqp.framing.definitions.BasicGetOk method), 34
get_size() (coolamqp.framing.definitions.BasicNack method), 34
get_size() (coolamqp.framing.definitions.BasicPublish method), 35
get_size() (coolamqp.framing.definitions.BasicQos method), 36
get_size() (coolamqp.framing.definitions.BasicRecover method), 38
get_size() (coolamqp.framing.definitions.BasicRecoverAsync method), 38
get_size() (coolamqp.framing.definitions.BasicReject method), 37
get_size() (coolamqp.framing.definitions.BasicReturn method), 37
get_size() (coolamqp.framing.definitions.ChannelClose method), 20
get_size() (coolamqp.framing.definitions.ChannelFlow method), 21
get_size() (coolamqp.framing.definitions.ChannelFlowOk method), 21
get_size() (coolamqp.framing.definitions.ConfirmSelect method), 39
get_size() (coolamqp.framing.definitions.ConnectionBlocked method), 15
get_size() (coolamqp.framing.definitions.ConnectionClose method), 15
get_size() (coolamqp.framing.definitions.ConnectionOpen method), 16
get_size() (coolamqp.framing.definitions.ConnectionSecure method), 17
get_size() (coolamqp.framing.definitions.ConnectionSecureOk method), 18
get_size() (coolamqp.framing.definitions.ConnectionStart method), 17
get_size() (coolamqp.framing.definitions.ConnectionStartOk method), 18
get_size() (coolamqp.framing.definitions.ConnectionTune method), 19
get_size() (coolamqp.framing.definitions.ConnectionTuneOk method), 20
get_size() (coolamqp.framing.definitions.ExchangeBind method), 22
get_size() (coolamqp.framing.definitions.ExchangeDeclare method), 23
get_size() (coolamqp.framing.definitions.ExchangeDelete method), 24
get_size() (coolamqp.framing.definitions.ExchangeUnbind method), 25
get_size() (coolamqp.framing.definitions.QueueBind method), 25
get_size() (coolamqp.framing.definitions.QueueDeclare method), 26
get_size() (coolamqp.framing.definitions.QueueDeclareOk method), 28
get_size() (coolamqp.framing.definitions.QueueDelete method), 27
get_size() (coolamqp.framing.definitions.QueueDeleteOk method), 28
get_size() (coolamqp.framing.definitions.QueuePurge method), 28
get_size() (coolamqp.framing.definitions.QueuePurgeOk method), 29
get_size() (coolamqp.framing.definitions.QueueUnbind method), 29

## H

HoldingFrameTracer (class in coolamqp.tracing), 46

## L

LoggingFrameTracer (*class in coolamqp.tracing*), 45

## M

Message (*class in coolamqp.objects*), 8, 44

## N

nack() (*coolamqp.objects.ReceivedMessage method*), 44
NodeDefinition (*class in coolamqp.objects*), 5, 43

## O

on_broker_cancel (*coolamqp.attaches.Consumer attribute*), 10
on_cancel (*coolamqp.attaches.Consumer attribute*), 10
on_close() (*coolamqp.attaches.Consumer method*), 10
on_delivery() (*coolamqp.attaches.Consumer method*), 10
on_frame() (*coolamqp.tracing.BaseFrameTracer method*), 45
on_frame() (*coolamqp.tracing.HoldingFrameTracer method*), 46
on_frame() (*coolamqp.tracing.LoggingFrameTracer method*), 45
on_operational() (*coolamqp.attaches.Consumer method*), 10
on_setup() (*coolamqp.attaches.Consumer method*), 11

## P

Properties (*coolamqp.objects.Message attribute*), 9
publish() (*coolamqp.clustering.Cluster method*), 2, 7, 42

## Q

QueueBind (*class in coolamqp.framing.definitions*), 25
QueueBindOk (*class in coolamqp.framing.definitions*), 26
QueueDeclare (*class in coolamqp.framing.definitions*), 26
QueueDeclareOk (*class in coolamqp.framing.definitions*), 27
QueueDelete (*class in coolamqp.framing.definitions*), 27
QueueDeleteOk (*class in coolamqp.framing.definitions*), 28
QueuePurge (*class in coolamqp.framing.definitions*), 28
QueuePurgeOk (*class in coolamqp.framing.definitions*), 29
QueueUnbind (*class in coolamqp.framing.definitions*), 29
QueueUnbindOk (*class in coolamqp.framing.definitions*), 30

## R

ReceivedMessage (*class in coolamqp.objects*), 44

## S

set_qos() (*coolamqp.attaches.Consumer method*), 11
shutdown() (*coolamqp.clustering.Cluster method*), 3, 7, 43
start() (*coolamqp.clustering.Cluster method*), 3, 8, 43

## T

TxCommit (*class in coolamqp.framing.definitions*), 38
TxCommitOk (*class in coolamqp.framing.definitions*), 39
TxRollback (*class in coolamqp.framing.definitions*), 39
TxRollbackOk (*class in coolamqp.framing.definitions*), 39
TxSelect (*class in coolamqp.framing.definitions*), 39
TxSelectOk (*class in coolamqp.framing.definitions*), 39

## W

write_arguments() (*coolamqp.framing.definitions.BasicAck method*), 30
write_arguments() (*coolamqp.framing.definitions.BasicCancel method*), 31
write_arguments() (*coolamqp.framing.definitions.BasicCancelOk method*), 32
write_arguments() (*coolamqp.framing.definitions.BasicConsume method*), 31
write_arguments() (*coolamqp.framing.definitions.BasicConsumeOk method*), 32
write_arguments() (*coolamqp.framing.definitions.BasicDeliver method*), 33
write_arguments() (*coolamqp.framing.definitions.BasicGet method*), 33
write_arguments() (*coolamqp.framing.definitions.BasicGetOk method*), 34
write_arguments() (*coolamqp.framing.definitions.BasicNack method*), 35
write_arguments() (*coolamqp.framing.definitions.BasicPublish method*), 35
write_arguments() (*coolamqp.framing.definitions.BasicQos method*), 36
write_arguments() (*coolamqp.framing.definitions.BasicRecover method*), 38
write_arguments() (*coolamqp.framing.definitions.BasicRecoverAsync method*), 38
write_arguments() (*coolamqp.framing.definitions.BasicReject method*), 37
write_arguments() (*coolamqp.framing.definitions.BasicReturn method*), 37
write_arguments() (*coolamqp.framing.definitions.ChannelClose method*), 20
write_arguments() (*coolamqp.framing.definitions.ChannelFlow method*), 21
write_arguments() (*coolamqp.framing.definitions.ChannelFlowOk method*), 22
write_arguments() (*coolamqp.framing.definitions.ConfirmSelect method*), 39

write_arguments() (*coolamqp.framing.definitions.ConnectionBlocked method*), 15
write_arguments() (*coolamqp.framing.definitions.ConnectionClose method*), 16
write_arguments() (*coolamqp.framing.definitions.ConnectionOpen method*), 16
write_arguments() (*coolamqp.framing.definitions.ConnectionSecure method*), 17
write_arguments() (*coolamqp.framing.definitions.ConnectionSecureOk method*), 19
write_arguments() (*coolamqp.framing.definitions.ConnectionStart method*), 17
write_arguments() (*coolamqp.framing.definitions.ConnectionStartOk method*), 18
write_arguments() (*coolamqp.framing.definitions.ConnectionTune method*), 19
write_arguments() (*coolamqp.framing.definitions.ConnectionTuneOk method*), 20
write_arguments() (*coolamqp.framing.definitions.ExchangeBind method*), 22
write_arguments() (*coolamqp.framing.definitions.ExchangeDeclare method*), 23
write_arguments() (*coolamqp.framing.definitions.ExchangeDelete method*), 24
write_arguments() (*coolamqp.framing.definitions.ExchangeUnbind method*), 25
write_arguments() (*coolamqp.framing.definitions.QueueBind method*), 26
write_arguments() (*coolamqp.framing.definitions.QueueDeclare method*), 27
write_arguments() (*coolamqp.framing.definitions.QueueDeclareOk method*), 28
write_arguments() (*coolamqp.framing.definitions.QueueDelete method*), 27
write_arguments() (*coolamqp.framing.definitions.QueueDeleteOk method*), 28
write_arguments() (*coolamqp.framing.definitions.QueuePurge method*), 29
write_arguments() (*coolamqp.framing.definitions.QueuePurgeOk method*), 29
write_arguments() (*coolamqp.framing.definitions.QueueUnbind method*), 30