amqplib

AMQP 0-9-1 library and client for Node.JS

Channel-oriented API reference

        - channel.sendToQueue
        - channel.consume
        - channel.cancel
        - channel.get
        - channel.ack
        - channel.ackAll
        - channel.nack
        - channel.nackAll
        - channel.reject
        - channel.prefetch
        - channel.recover
    - ConfirmChannel
        - confirmChannel.waitForConfirms
- RabbitMQ and deletion

There are two parallel client APIs available. One uses promises, and the other uses callbacks, *mutatis mutandis*. Since they present much the same set of objects and methods, you are free to choose which you prefer, without missing out on any features. However, they don't mix -- a channel from the promises API has only the promise-based methods, and likewise the callback API -- so it is best to pick one and stick with it.

The promise-based API is the "main" module in the library:

```
var amqp = require('amqplib');
```

You can access the callback-based API this way:

```
var amqp = require('amqplib/callback_api');
```

In the following I use "resolve", "resolving" etc., to refer either to resolving a returned promise (usually with a value); or in the callback API, invoking a supplied callback with `null` as the first argument (and usually some value as the second argument). Likewise, "reject" etc., will mean rejecting a promise or calling the callback with an `Error` as the first argument (and no value).

Overview

The client APIs are based closely on the protocol model. The general idea is to connect, then create one or more channels on which to issue commands, send messages, and so on. Most errors in AMQP invalidate just the channel which had problems, so this ends up being a fairly natural way to use AMQP. The downside is that it doesn't give any guidance on *useful* ways to use AMQP; that is, it does little beyond giving access to the various AMQP commands.

Most operations in AMQP are RPCs, synchronous at the channel layer of the protocol but asynchronous from the library's point of view. Accordingly, most methods either return promises, or accept callbacks, yielding the server's reply (often containing useful information such as generated identifiers). RPCs are queued by the channel if it is already waiting for a reply -- synchronising on RPCs in this way is implicitly required by the protocol specification.

Some methods are not RPCs -- they do not have responses from the server. These return either nothing (`ack[All]`, `nack[All]`, `reject`) or a boolean (`publish` and `sendToQueue`); see flow control.

Dealing with failure

Most operations in AMQP act like assertions, failing if the desired conditions cannot be met; for example, if a queue being declared already exists but with different properties. A failed operation will

- reject the current RPC, if there is one
- invalidate the channel object, meaning further operations will throw an exception
- reject any RPCs waiting to be sent
- cause the channel object to emit `'error'`
- cause the channel object to emit `'close'`

Since the RPCs are effectively synchronised, any such channel error is very likely to have been caused by the outstanding RPC. However, it's often sufficient to fire off a number of RPCs and check only the result for the last, since it'll be rejected if it or any of its predecessors fail.

The exception thrown on operations subsequent to a failure *or* channel close also contains the stack at the point that the channel was closed, in the field `stackAtStateChange`. This may be useful to determine what has caused an unexpected closure.

```
connection.createChannel().then(function(ch) {
  ch.close();
  try {
    ch.close();
  }
  catch (alreadyClosed) {
    console.log(alreadyClosed.stackAtStateChange);
  }
});
```

Exceptions and promises

Promises returned from methods are amenable to composition using, for example, when.js's functions:

```
amqp.connect().then(function(conn) {
  var ok = conn.createChannel();
  ok = ok.then(function(ch) {
    return when.all([
      ch.assertQueue('foo'),
      ch.assertExchange('bar'),
      ch.bindQueue('foo', 'bar', 'baz'),
      ch.consume('foo', handleMessage)
    ]);
  });
  return ok;
}).then(null, console.warn);
```

If an exception is thrown in a promise continuation, the promise library will redirect control to a *following* error continuation:

```
amqp.connect().then(function(conn) {
 // Everything ok, but ..
 throw new Error('SNAFU');
}, function(err) {
 console.error('Connect failed: %s', err);
}).then(null, function(err) {
  console.error('Connect succeeded, but error thrown: %s', err);
});
```

Exceptions and callbacks

The callback API expects callbacks that follow the convention `function(err, value) {...}`. This library does not attempt to deal with exceptions thrown in callbacks, so in general they will trigger the last-resort `'uncaughtException'` event of the process.

However, since the connection and channels are `EventEmitter`s, they can be bound to a domain:

```
var dom = domain.create();
dom.on('error', gracefullyRestart);

amqp.connect(function(err, conn) {
 dom.add(conn);
 //...
});
```

Implicit binding works for connections or channels created within a `Domain#run`.

```
var dom = domain.create();
dom.on('error', gracefullyRestart);

dom.run(function() {
  amqp.connect(function(err, conn) {
      // ...
  });
});
```

Flow control

Channels act like `stream.Writable` when you call `publish` or `sendToQueue`: they return either `true`, meaning "keep sending", or `false`, meaning "please wait for a 'drain' event".

Those methods, along with `ack`, `ackAll`, `nack`, `nackAll`, and `reject`, do not have responses from the server. This means they *do not return a promise* in the promises API. The `ConfirmChannel` *does* accept a callback in *both* APIs, called when the server confirms the message; as well as returning a boolean.

Argument handling

Many operations have mandatory arguments as well as optional arguments with defaults; in general, the former appear as parameters to the method while latter are collected in a single `options` parameter, to be supplied as an object with the fields mentioned. Extraneous fields in `options` are ignored, so it is often possible to coalesce the options for a number of operations into a single object, should that be convenient. Likewise, fields from the prototype chain are accepted, so a common `options` value can be specialised by e.g., using `Object.create(common)` then setting some fields.

Often, AMQP commands have an `arguments` table that can contain arbitrary values, usually used by implementation-specific extensions like RabbitMQ's consumer priorities. This is accessible as the option `arguments`, also an object: if an API method does not account for an extension in its stated `options`, you can fall back to using the `options.arguments` object, though bear in mind that the field name will usually be 'x-something', while the options are just 'something'. Values passed in `options`, if understood by the API, will override those given in `options.arguments`.

```
var common_options = {durable: true, noAck: true};
ch.assertQueue('foo', common_options);
// Only 'durable' counts for queues

var bar_opts = Object.create(common_options);
bar_opts.autoDelete = true;
// "Subclass" our options
ch.assertQueue('bar', bar_opts);

var foo_consume_opts = Object.create(common_options);
foo_consume_opts.arguments = {'x-priority': 10};
ch.consume('foo', console.log, foo_consume_opts);
// Use the arguments table to give a priority, even though it's
// available as an option

var bar_consume_opts = Object.create(foo_consume_opts);
bar_consume_opts.priority = 5;
ch.consume('bar', console.log, bar_consume_opts);
// The 'priority' option will override that given in the arguments
// table
```

API reference

connect

Promises

```
connect([url, [socketOptions]])
```

Callbacks

```
connect([url, [socketOptions]], function(err, conn) {...})
```

Connect to an AMQP 0-9-1 server, optionally given an AMQP URL (see AMQP URI syntax) and socket options. The protocol part (`amqp:` or `amqps:`) is mandatory; defaults for elided parts are as given in `'amqp://guest:guest@localhost:5672'`. If the URI is omitted entirely, it will default to `'amqp://localhost'`, which given the defaults for missing parts, will connect to a RabbitMQ installation with factory settings, on localhost.

For convenience, an *absent* path segment (e.g., as in the URLs just given) is interpreted as the virtual host named `/`, which is present in RabbitMQ out of the box. Per the URI specification, *just a trailing slash* as in `'amqp://localhost/'` would indicate the virtual host with an empty name, which does not exist unless it's been explicitly created. When specifying another virtual host, remember that its name must be escaped; so e.g., the virtual host named `/foo` is `'%2Ffoo'`; in a full URI, `'amqp://localhost/%2Ffoo'`.

Further AMQP tuning parameters may be given in the query part of the URI, e.g., as in `'amqp://localhost?frameMax=0x1000'`. These are:

- `frameMax`, the size in bytes of the maximum frame allowed over the connection. `0` means no limit (but since frames have a size field which is an unsigned 32 bit integer, it's perforce `2^32 - 1`); I default it to 0x1000, i.e. 4kb, which is the allowed minimum, will fit many purposes, and not chug through Node.JS's buffer pooling.

- `channelMax`, the maximum number of channels allowed. Default is `0`, meaning `2^16 - 1`.

- `heartbeat`: the period of the connection heartbeat, in seconds. Defaults to `0`, meaning no heartbeat. OMG no heartbeat!

- `locale`: the desired locale for error messages, I suppose. RabbitMQ only ever uses `en_US`; which, happily, is the default.

The socket options will be passed to the socket library (`net` or `tls`). In an exception to the general rule, *they must be fields set on the object supplied*; that is, not in the prototype chain. The socket options is useful for supplying certificates and so on for an SSL connection; see the SSL guide.

The socket options may also include the key `noDelay`, with a boolean value. If the value is `true`, this sets `TCP_NODELAY` on the underlying socket.

The returned promise, or supplied callback, will either be resolved with an object representing an open connection, or rejected with a sympathetically-worded error (in en_US).

Supplying a malformed URI will cause `connect()` to throw an exception; other problems, including refused and dropped TCP connections, will result in a rejection.

RabbitMQ since version 3.2.0 will send a frame to notify the client of authentication failures, which results in a rejection. RabbitMQ before version 3.2.0, per the AMQP specification, will close the socket in the case of an authentication failure, making a dropped connection ambiguous (it will also wait a few seconds before doing so).

Heartbeating

If you supply a non-zero period in seconds as the `heartbeat` parameter, the connection will be monitored for liveness. If the client fails to read data from the connection for two successive intervals, the connection will emit an error and close. It will also send heartbeats to the server (in the absence of other data).

ChannelModel and CallbackModel

These constructors represent connections in the channel APIs. They take as an argument a `connection.Connection`. It is better to use `connect()`, which will open the connection for you. The constructors are exported as potential extension points.

{Channel,Callback}Model#close

Promises

```
connection.close()
```

Callbacks

```
connection.close([function(err) {...}])
```

Close the connection cleanly. Will immediately invalidate any unresolved operations, so it's best to make sure you've done everything you need to before calling this. Will be resolved once the connection, and underlying socket, are closed. The model will also emit `'close'` at that point.

Although it's not strictly necessary, it will avoid some warnings in the server log if you close the connection before exiting:

```
var open = amqp.connect();
open.then(function(conn) {
  var ok = doStuffWithConnection(conn);
  return ok.then(conn.close.bind(conn));
}).then(null, console.warn);
```

Note that I'm synchronising on the return value of `doStuffWithConnection()`, assumed here to be a promise, so that I can be sure I'm all done. The callback version looks like this:

```
amqp.connect(function(err, conn) {
  if (err !== null) return console.warn(err);
  doStuffWithConnection(conn, function() {
    conn.close();
  });
});
```

There it's assumed that doStuffWithConnection invokes its second argument once it's all finished.

If your program runs until interrupted, you can hook into the process signal handling to close the connection:

```
var open = amqp.connect();
open.then(function(conn) {
  process.once('SIGINT', conn.close.bind(conn));
  return doStuffWithConnection(conn);
}).then(null, console.warn);
```

NB it's no good using `process.on('exit', ...)`, since `close()` needs to do I/O.

{Channel,Callback}Model events

`#on('close', function() {...})`

Emitted once the closing handshake initiated by `#close()` has completed; or, if server closed the connection, once the client has sent the closing handshake; or, if the underlying stream (e.g., socket) has closed.

In the case of a server-initiated shutdown *or* an error, the `'close'` handler will be supplied with an error indicating the cause. You can ignore this if you don't care why the connection closed; or, you can test it with `require('amqplib/lib/connection').isFatalError(err)` to see if it was a crash-worthy error.

`#on('error', function (err) {...})`

Emitted if the connection closes for a reason other than `#close` being called or a graceful server-initiated close; such reasons include:

- a protocol transgression the server detected (likely a bug in this library)
- a server error
- a network error
- the server thinks the client is dead due to a missed heartbeat

A graceful close may be initiated by an operator (e.g., with an admin tool), or if the server is shutting down; in this case, no `'error'` event will be emitted.

`'close'` will also be emitted, after `'error'`.

`#on('blocked', function(reason) {...})`

Emitted when a RabbitMQ server (after version 3.2.0) decides to block the connection. Typically it will do this if there is some resource shortage, e.g., memory, and messages are published on the connection. See the RabbitMQ documentation for this extension for details.

```
#on('unblocked', function() {...})
```

Emitted at some time after `'blocked'`, once the resource shortage has alleviated.

{Channel,Callback}Model#createChannel

Promises

```
#createChannel()
```

Callbacks

```
#createChannel(function(err, channel) {...})
```

Resolves to an open `Channel` (The callback version returns the channel; but it is not usable before the callback has been invoked). May fail if there are no more channels available (i.e., if there are already `channelMax` channels open).

{Channel,Callback}Model#createConfirmChannel

Promises

```
#createConfirmChannel()
```

Callbacks

```
#createConfirmChannel(function(err, channel) {...})
```

Open a fresh channel, switched to "confirmation mode". See `ConfirmChannel` below.

Channels

There are channel objects in each of the APIs, and these contain most of the methods for getting things done.

```
new Channel(connection)
```

This constructor represents a protocol channel. Channels are multiplexed over connections, and represent something like a session, in that most operations (and thereby most errors) are scoped to channels.

The constructor is exported from the API modules as an extension point. When using a client API, obtain an open `Channel` by opening a connection (`connect()` above) and calling `#createChannel` or `#createConfirmChannel`.

Channel#close

Promises

```
Channel#close()
```

Callbacks

```
Channel#close([function(err) {...}])
```

Close a channel. Will be resolved with no value once the closing handshake is complete.

There's not usually any reason to close a channel rather than continuing to use it until you're ready to close the connection altogether. However, the lifetimes of consumers are scoped to channels, and thereby other things such as exclusive locks on queues, so it is occasionally worth being deliberate about opening and closing channels.

Channel events

```
#on('close', function() {...})
```

A channel will emit `'close'` once the closing handshake (possibly initiated by `#close()`) has completed; or, if its connection closes.

When a channel closes, any unresolved operations on the channel will be abandoned (and the returned promises rejected).

`#on('error', function(err) {...})`

A channel will emit `'error'` if the server closes the channel for any reason. Such reasons include

- an operation failed due to a failed precondition (usually something named in an argument not existing)
- an human closed the channel with an admin tool

A channel will not emit `'error'` if its connection closes with an error.

`#on('return', function(msg) {...})`

If a message is published with the `mandatory` flag (it's an option to `Channel#publish` in this API), it may be returned to the sending channel if it cannot be routed. Whenever this happens, the channel will emit `return` with a message object (as described in `#consume`) as an argument.

`#on('drain', function() {...})`

Like a stream.Writable, a channel will emit `'drain'`, if it has previously returned `false` from `#publish` or `#sendToQueue`, once its write buffer has been emptied (i.e., once it is ready for writes again).

Channel#assertQueue

Promises

`#assertQueue([queue, [options]])`

Callbacks

`#assertQueue([queue, [options, [function(err, ok) {...}]]])`

Assert a queue into existence. This operation is idempotent given identical arguments; however, it will bork the channel if the queue already exists but has different properties (values supplied in the `arguments` field may or may not count for borking purposes; check the borker's, I mean broker's, documentation).

`queue` is a string; if you supply an empty string or other falsey value (including `null` and `undefined`), the server will create a random name for you.

`options` is an object and may be empty or null, or outright omitted if it's the last argument. The relevant fields in options are:

- `exclusive`: if true, scopes the queue to the connection (defaults to false)
- `durable`: if true, the queue will survive broker restarts, modulo the effects of `exclusive` and `autoDelete`; this defaults to true if not supplied, unlike the others
- `autoDelete`: if true, the queue will be deleted when the number of consumers drops to zero (defaults to false)
- `arguments`: additional arguments, usually parameters for some kind of broker-specific extension e.g., high availability, TTL.

RabbitMQ extensions can also be supplied as options. These typically require non-standard `x-*` keys and values, sent in the `arguments` table; e.g., `'x-expires'`. When supplied in `options`, the `x-` prefix for the key is removed; e.g., `'expires'`. Values supplied in `options` will overwrite any analogous field you put in `options.arguments`.

- `messageTtl` (0 <= n < 2^32): expires messages arriving in the queue after n milliseconds
- `expires` (0 < n < 2^32): the queue will be destroyed after n milliseconds of disuse, where use means having consumers, being

declared (asserted or checked, in this API), or being polled with a `#get`.

- `deadLetterExchange` (string): an exchange to which messages discarded from the queue will be resent. Use `deadLetterRoutingKey` to set a routing key for discarded messages; otherwise, the message's routing key (and CC and BCC, if present) will be preserved. A message is discarded when it expires or is rejected or nacked, or the queue limit is reached.

- `maxLength` (positive integer): sets a maximum number of messages the queue will hold. Old messages will be discarded (dead-lettered if that's set) to make way for new messages.

- `maxPriority` (positive integer): makes the queue a priority queue.

Resolves to the "ok" reply from the server, which includes fields for the queue name (important if you let the server name it), a recent consumer count, and a recent message count; e.g.,

```
{
  queue: 'foobar',
  messageCount: 0,
  consumerCount: 0
}
```

Channel#checkQueue

Promises

```
#checkQueue(queue)
```

Callbacks

```
#checkQueue(queue, [function(err, ok) {...}])
```

Check whether a queue exists. This will bork the channel if the named queue *doesn't* exist; if it does exist, you go through to the next round!

There's no options, unlike `#assertQueue()`, just the queue name. The reply from the server is the same as for `#assertQueue()`.

Channel#deleteQueue

Promises

```
#deleteQueue(queue, [options])
```

Callbacks

```
#deleteQueue(queue, [options, [function(err, ok) {...}]])
```

Delete the queue named. Naming a queue that doesn't exist will result in the server closing the channel, to teach you a lesson (except in RabbitMQ version 3.2.0 and after[1]). The options here are:

- `ifUnused` (boolean): if true and the queue has consumers, it will not be deleted and the channel will be closed. Defaults to false.
- `ifEmpty` (boolean): if true and the queue contains messages, the queue will not be deleted and the channel will be closed. Defaults to false.

Note the obverse semantics of the options: if both are true, the queue will be deleted only if it has no consumers *and* no messages.

You should leave out the options altogether if you want to delete the queue unconditionally.

The server reply contains a single field, `messageCount`, with the number of messages deleted or dead-lettered along with the queue.

Channel#purgeQueue

Promises

```
#purgeQueue(queue)
```

Callbacks

`#purgeQueue(queue, [function(err, ok) {...}])`

Remove all undelivered messages from the `queue` named. Note that this won't remove messages that have been delivered but not yet acknowledged; they will remain, and may be requeued under some circumstances (e.g., if the channel to which they were delivered closes without acknowledging them).

The server reply contains a single field, `messageCount`, containing the number of messages purged from the queue.

Channel#bindQueue

Promises

`#bindQueue(queue, source, pattern, [args])`

Callbacks

`#bindQueue(queue, source, pattern, [args, [function(err, ok) {...}]])`

Assert a routing path from an exchange to a queue: the exchange named by `source` will relay messages to the `queue` named, according to the type of the exchange and the `pattern` given. The RabbitMQ tutorials give a good account of how routing works in AMQP.

`args` is an object containing extra arguments that may be required for the particular exchange type (for which, see your server's documentation). It may be omitted if it's the last argument, which is equivalent to an empty object.

The server reply has no fields.

Channel#unbindQueue

Promises

`#unbindQueue(queue, source, pattern, [args])`

Callbacks

`#unbindQueue(queue, source, pattern, [args, [function(err, ok) {...}]])`

Remove a routing path between the `queue` named and the exchange named as `source` with the `pattern` and arguments given. Omitting `args` is equivalent to supplying an empty object (no arguments). Beware: attempting to unbind when there is no such binding may result in a punitive error (the AMQP specification says it's a connection-killing mistake; RabbitMQ before version 3.2.0 softens this to a channel error, and from version 3.2.0, doesn't treat it as an error at all[1]. Good ol' RabbitMQ).

Channel#assertExchange

Promises

`#assertExchange(exchange, type, [options])`

Callbacks

`#assertExchange(exchange, type, [options, [function(err, ok) {...}]])`

Assert an exchange into existence. As with queues, if the exchange exists already and has properties different to those supplied, the channel will 'splode; fields in the arguments object may or may not be 'splodey, depending on the type of exchange. Unlike queues, you must supply a name, and it can't be the empty string. You must also supply an exchange type, which determines how messages will be routed through the exchange.

NB There is just one RabbitMQ extension pertaining to exchanges in general (`alternateExchange`); however, specific exchange types may use the `arguments` table to supply parameters.

The options:

- `durable` (boolean): if true, the exchange will survive broker restarts. Defaults to true.
- `internal` (boolean): if true, messages cannot be published directly to the exchange (i.e., it can only be the target of bindings, or possibly create messages ex-nihilo). Defaults to false.
- `autoDelete` (boolean): if true, the exchange will be destroyed once the number of bindings for which it is the source drop to zero. Defaults to false.
- `alternateExchange` (string): an exchange to send messages to if this exchange can't route them to any queues.
- `arguments` (object): any additional arguments that may be needed by an exchange type.

The server reply echoes the exchange name, in the field `exchange`.

Channel#checkExchange

Promises

```
#checkExchange(exchange)
```

Callbacks

```
#checkExchange(exchange, [function(err, ok) {...}])
```

Check that an exchange exists. If it doesn't exist, the channel will be closed with an error. If it does exist, happy days.

Channel#deleteExchange

Promises

`#deleteExchange(name, [options])`

Callbacks

`#deleteExchange(name, [options, [function(err, ok) {...}]])`

Delete an exchange. The only meaningful field in `options` is:

- `ifUnused` (boolean): if true and the exchange has bindings, it will not be deleted and the channel will be closed.

If the exchange does not exist, a channel error is raised (RabbitMQ version 3.2.0 and after will not raise an error[1]).

The server reply has no fields.

Channel#bindExchange

Promises

`#bindExchange(destination, source, pattern, [args])`

Callbacks

`#bindExchange(destination, source, pattern, [args, [function(err, ok) {...}]])`

Bind an exchange to another exchange. The exchange named by `destination` will receive messages from the exchange named by `source`, according to the type of the source and the `pattern` given. For example, a `direct` exchange will relay messages that have a routing key equal to the pattern.

NB Exchange to exchange binding is a RabbitMQ extension.

The server reply has no fields.

Channel#unbindExchange

Promises

```
#unbindExchange(destination, source, pattern, [args])
```

Callbacks

```
#unbindExchange(destination, source, pattern, [args,
[function(err, ok) {...}]])
```

Remove a binding from an exchange to another exchange. A binding with the exact `source` exchange, `destination` exchange, routing key `pattern`, and extension `args` will be removed. If no such binding exists, it's – you guessed it – a channel error, except in RabbitMQ >= version 3.2.0, for which it succeeds trivially[1].

Channel#publish

Promises or callbacks

```
#publish(exchange, routingKey, content, [options])
```

Publish a single message to an exchange. The mandatory parameters are:

- `exchange` and `routingKey`: the exchange and routing key, which determine where the message goes. A special case is sending `''` as the exchange, which will send directly to the queue named by the routing key; `#sendToQueue` below is equivalent to this special case. If the named exchange does not exist, the channel will be closed.

- `content`: a buffer containing the message content. This will be copied during encoding, so it is safe to mutate it once this method has returned.

The remaining parameters are provided as fields in `options`, and are divided into those that have some meaning to RabbitMQ and those that will be ignored by RabbitMQ but passed on to consumers. `options` may be omitted altogether, in which case defaults as noted will apply.

The "meaningful" options are a mix of fields in BasicDeliver (the method used to publish a message), BasicProperties (in the message header frame) and RabbitMQ extensions which are given in the `headers` table in BasicProperties.

Used by RabbitMQ and sent on to consumers:

- `expiration` (string): if supplied, the message will be discarded from a queue once it's been there longer than the given number of milliseconds. In the specification this is a string; numbers supplied here will be coerced to strings for transit.

- `userId` (string): If supplied, RabbitMQ will compare it to the username supplied when opening the connection, and reject messages for which it does not match.

- `CC` (string or array of string): an array of routing keys as strings; messages will be routed to these routing keys in addition to that given as the `routingKey` parameter. A string will be implicitly treated as an array containing just that string. This will override any value given for `CC` in the `headers` parameter. **NB** The property names `CC` and `BCC` are case-sensitive.

- `priority` (positive integer): a priority for the message; ignored by versions of RabbitMQ older than 3.5.0, or if the queue is not a priority queue (see `maxPriority` above).

- `persistent` (boolean): If truthy, the message will survive broker restarts provided it's in a queue that also survives restarts. Corresponds to, and overrides, the property `deliveryMode`.

- `deliveryMode` (boolean or numeric): Either `1` or falsey, meaning non-persistent; or, `2` or truthy, meaning persistent. That's just obscure though. Use the option `persistent` instead.

Used by RabbitMQ but not sent on to consumers:

- `mandatory` (boolean): if true, the message will be returned if it is not routed to a queue (i.e., if there are no bindings that match its routing key).
- `BCC` (string or array of string): like `CC`, except that the value will not be sent in the message headers to consumers.

Not used by RabbitMQ and not sent to consumers:

- `immediate` (boolean): in the specification, this instructs the server to return the message if it is not able to be sent immediately to a consumer. No longer implemented in RabbitMQ, and if true, will provoke a channel error, so it's best to leave it out.

Ignored by RabbitMQ (but may be useful for applications):

- `contentType` (string): a MIME type for the message content
- `contentEncoding` (string): a MIME encoding for the message content
- `headers` (object): application specific headers to be carried along with the message content. The value as sent may be augmented by extension-specific fields if they are given in the parameters, for example, 'CC', since these are encoded as message headers; the supplied value won't be mutated
- `correlationId` (string): usually used to match replies to requests, or similar
- `replyTo` (string): often used to name a queue to which the receiving application must send replies, in an RPC scenario (many libraries assume this pattern)

- `messageId` (string): arbitrary application-specific identifier for the message
- `timestamp` (positive number): a timestamp for the message
- `type` (string): an arbitrary application-specific type for the message
- `appId` (string): an arbitrary identifier for the originating application

`#publish` mimics the [`stream.Writable`]() interface in its return value; it will return `false` if the channel's write buffer is 'full', and `true` otherwise. If it returns `false`, it will emit a `'drain'` event at some later time.

Channel#sendToQueue

Promises and callbacks

`#sendToQueue(queue, content, [options])`

Send a single message with the `content` given as a buffer to the specific `queue` named, bypassing routing. The options and return value are exactly the same as for `#publish`.

Channel#consume

Promises

`#consume(queue, function(msg) {...}, [options])`

Callbacks

`#consume(queue, function(msg) {...}, [options, [function(err, ok) {...}]])`

Set up a consumer with a callback to be invoked with each message.

Options (which may be omitted if the last argument):

- `consumerTag` (string): a name which the server will use to distinguish message deliveries for the consumer; mustn't be already in use on the channel. It's usually easier to omit this, in which case the server will create a random name and supply it in the reply.

- `noLocal` (boolean): in theory, if true then the broker won't deliver messages to the consumer if they were also published on this connection; RabbitMQ doesn't implement it though, and will ignore it. Defaults to false.

- `noAck` (boolean): if true, the broker won't expect an acknowledgement of messages delivered to this consumer; i.e., it will dequeue messages as soon as they've been sent down the wire. Defaults to false (i.e., you will be expected to acknowledge messages).

- `exclusive` (boolean): if true, the broker won't let anyone else consume from this queue; if there already is a consumer, there goes your channel (so usually only useful if you've made a 'private' queue by letting the server choose its name).

- `priority` (integer): gives a priority to the consumer; higher priority consumers get messages in preference to lower priority consumers. See this RabbitMQ extension's documentation

- `arguments` (object): arbitrary arguments. Go to town.

The server reply contains one field, `consumerTag`. It is necessary to remember this somewhere if you will later want to cancel this consume operation (i.e., to stop getting messages).

The message callback supplied in the second argument will be invoked with message objects of this shape:

```
{
  content: Buffer,
  fields: Object,
  properties: Object
}
```

The message `content` is a buffer containing the bytes published.

The fields object has a handful of bookkeeping values largely of interest only to the library code: `deliveryTag`, a serial number for the message; `consumerTag`, identifying the consumer for which the message is destined; `exchange` and `routingKey` giving the routing information with which the message was published; and, `redelivered`, which if true indicates that this message has been delivered before and been handed back to the server (e.g., by a nack or recover operation).

The `properties` object contains message properties, which are all the things mentioned under `#publish` as options that are transmitted. Note that RabbitMQ extensions (just `CC`, presently) are sent in the `headers` table so will appear there in deliveries.

If the consumer is cancelled by RabbitMQ, the message callback will be invoked with `null`.

Channel#cancel

Promises

```
#cancel(consumerTag)
```

Callbacks

```
#cancel(consumerTag, [function(err, ok) {...}])
```

This instructs the server to stop sending messages to the consumer identified by `consumerTag`. Messages may arrive between sending this and getting its reply; once the reply has resolved, however, there will be no more messages for the consumer, i.e., the message callback will no longer be invoked.

The `consumerTag` is the string given in the reply to `#consume`, which may have been generated by the server.

Channel#get

Promises

```
#get(queue, [options])
```

Callbacks

```
#get(queue, [options, [function(err, msgOrFalse) {...}]])
```

Ask a queue for a message, as an RPC. This will be resolved with either `false`, if there is no message to be had (the queue has no messages ready), or a message in the same shape as detailed in `#consume`.

Options:

- `noAck` (boolean): if true, the message will be assumed by the server to be acknowledged (i.e., dequeued) as soon as it's been sent over the wire. Default is false, that is, you will be expected to acknowledge the message.

Channel#ack

Promises and callbacks

```
#ack(message, [allUpTo])
```

Acknowledge the given message, or all messages up to and including the given message.

If a `#consume` or `#get` is issued with noAck: false (the default), the server will expect acknowledgements for messages before forgetting about them. If no such acknowledgement is given, those messages may be requeued once the channel is closed.

If `allUpTo` is true, all outstanding messages prior to and including the given message shall be considered acknowledged. If false, or omitted, only the message supplied is acknowledged.

It's an error to supply a message that either doesn't require acknowledgement, or has already been acknowledged. Doing so will errorise the channel. If you want to acknowledge all the messages and you don't have a specific message around, use `#ackAll`.

Channel#ackAll

Promises and callbacks

```
#ackAll()
```

Acknowledge all outstanding messages on the channel. This is a "safe" operation, in that it won't result in an error even if there are no such messages.

Channel#nack

Promises and callbacks

```
#nack(message, [allUpTo, [requeue]])
```

Reject a message. This instructs the server to either requeue the message or throw it away (which may result in it being dead-lettered).

If `allUpTo` is truthy, all outstanding messages prior to and including the given message are rejected. As with `#ack`, it's a channel-ganking error to use a message that is not outstanding. Defaults to `false`.

If `requeue` is truthy, the server will try to put the message or messages back on the queue or queues from which they came. Defaults to `true` if not given, so if you want to make sure messages are dead-lettered or discarded, supply `false` here.

This and `#nackAll` use a RabbitMQ-specific extension.

Channel#nackAll

Promises and callbacks

`#nackAll([requeue])`

Reject all messages outstanding on this channel. If `requeue` is truthy, or omitted, the server will try to re-enqueue the messages.

Channel#reject

Promises and callbacks

`#reject(message, [requeue])`

Reject a message. Equivalent to `#nack(message, false, requeue)`, but works in older versions of RabbitMQ (< v2.3.0) where `#nack` does not.

Channel#prefetch

Promises and callbacks

`#prefetch(count, [global])`

Set the prefetch count for this channel. The `count` given is the maximum number of messages sent over the channel that can be awaiting acknowledgement; once there are `count` messages outstanding, the server will not send more messages on this channel until one or more have been acknowledged. A falsey value for `count` indicates no such limit.

NB RabbitMQ v3.3.0 changes the meaning of prefetch (basic.qos) to apply per-*consumer*, rather than per-channel. It will apply to consumers started after the method is called. See rabbitmq-prefetch.

Use the `global` flag to get the per-channel behaviour. To keep life interesting, using the `global` flag with an RabbitMQ older than v3.3.0 will bring down the whole connection.

Channel#recover

Promises

```
#recover()
```

Callbacks

```
#recover([function(err, ok) {...}])
```

Requeue unacknowledged messages on this channel. The server will reply (with an empty object) once all messages are requeued.

ConfirmChannel

A channel which uses "confirmation mode" (a RabbitMQ extension).

On a channel in confirmation mode, each published message is 'acked' or (in exceptional circumstances) 'nacked' by the server, thereby indicating that it's been dealt with.

A confirm channel has the same methods as a regular channel, except that `#publish` and `#sendToQueue` accept a callback as an additional argument:

```
var open = require('amqplib').connect();
open.then(function(c) {
  c.createConfirmChannel().then(function(ch) {
    ch.sendToQueue('foo', new Buffer('foobar'), {},
                   function(err, ok) {
                     if (err !== null)
                       console.warn('Message nacked!');
                     else
                       console.log('Message acked');
    });
  });
});
```

Or, with the callback API:

```
require('amqplib/callback_api').connect(function(err, c) {
  c.createConfirmChannel(function(err, ch) {
    ch.sendToQueue('foo', new Buffer('foobar'), {}, function(err, ok)
      if (err !== null) console.warn('Message nacked!');
      else console.log('Message acked');
    });
  });
});
```

In practice this means the `options` argument must be supplied, at least as an empty object.

There are, broadly speaking, two uses for confirms. The first is to be able to act on the information that a message has been accepted, for example by responding to an upstream request. The second is to rate limit a publisher by limiting the number of unconfirmed messages it's allowed.

```
new ConfirmChannel(connection)
```

This constructor is a channel that uses confirms. It is exported as an extension point. To obtain such a channel, use `connect` to get a connection, then call `#createConfirmChannel`.

ConfirmChannel#waitForConfirms

Promises

```
#waitForConfirms()
```

Callbacks

```
#waitForConfirms(function(err) {...})
```

Resolves the promise, or invokes the callback, when all published messages have been confirmed. If any of the messages has been nacked, this will result in an error; otherwise the result is no value. Either way, the channel is still usable afterwards. It is also possible to call waitForConfirms multiple times without waiting for previous invocations to complete.

RabbitMQ and deletion

RabbitMQ version 3.2.0 makes queue and exchange deletions (and unbind) effectively idempotent, by not raising an error if the exchange, queue, or binding does not exist.

This does not apply to preconditions given to the operations. For example deleting a queue with `{ifEmpty: true}` will still fail if there are messages in the queue.