Channel:

Before you consume from or publish to Rabbit, you first have to connect to it. By

connecting, you’re creating a TCP connection between your app and the Rabbit broker.

Once the TCP connection is open (and you’re authenticated), your app then creates

an AMQP channel. This channel is a virtual connection inside the “real” TCP

connection, and it’s over the channel that you issue AMQP commands. Every channel

has a unique ID assigned to it.

Why do we need channels?

The main reason is because setting up and tearing down TCP sessions is expensive for an

operating system.

Exchanges:

There are four:

direct, fanout, topic, and headers. Each implements a different routing algorithm.

Headers.: allows you to match against a header in the AMQP message

instead of the routing key. Other than that, it operates identically to the direct

exchange but with much worse performance.( As a result, it doesn’t provide much

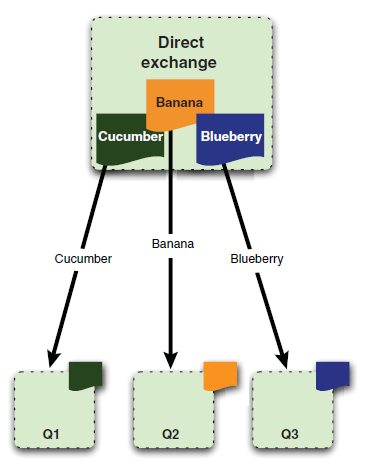
real-world benefit and is almost never

used.)

Direct: if the routing key matches, then the message

is delivered to the corresponding queue. The broker must implement the direct exchange, including a default exchange with an empty string as its name. When a queue is declared, it’ll be automatically

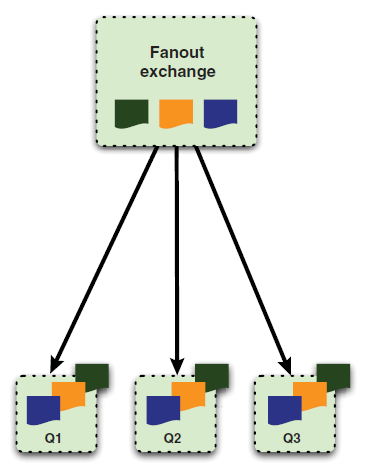
bound to that exchange using the queue name as routing key.



Fanout: this exchange will multicast the received message to the bound

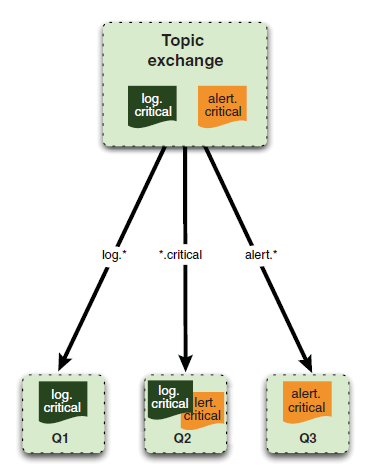
queues. when you send a message to a fanout exchange, it’ll be delivered to all the queues attached

to this exchange.



Topic: This exchange allows you to achieve interesting messaging scenarios, where messages can arrive to the same

queue coming from different sources



Let’s recap what you’ve learned in this section:

 The key components in the AMQP architecture are exchanges, queues, and

bindings.

 You bind queues to exchanges based on binding rules.

 Messages are sent to exchanges.

 There are three exchange types: direct, fanout, and topic.

 Based on the message routing key and the exchange type, the broker will

decide to which queue it has to deliver the message.

***virtual hosts***

Within every RabbitMQ server is the ability to create virtual message brokers called *virtual*

*hosts* (*vhosts*). Each one is essentially a mini-RabbitMQ server with its own queues,

exchanges, and bindings … and, more important, its own permissions.

This lets you safely use one RabbitMQ server for multiple applications without worrying that your

Sudoku app might delete queues used by your lost Fido tracker. This is useful for anything from

separating multiple customers on the same Rabbit to avoiding naming collisions on queues

and exchanges.

Vhosts are so fundamental to the concept of AMQP that you have to specify one

when you connect. RabbitMQ makes it easy to get started by including a default vhost

called / right out of the box. If you don’t need multiple vhosts, just use the default one.

It’s accessible using the default *guest* username with password *guest*, though you should

change the password for security

When you create a user in Rabbit, it’s usually assigned to at least one vhost and will

only be able to access queues, exchanges, and bindings on those assigned vhosts. Also,

when you’re designing your messaging architecture, keep in mind that separation

between vhosts is absolute. You can’t bind an exchange on vhost banana\_tree to a

queue on vhost oak\_tree.

If you need to find out what vhosts are running on a particular Rabbit server, run

$ rabbitmqctl list\_vhosts

***Durability***

Queues and exchanges in Rabbit: by default they don’t survive reboot.

On every queue and exchange the default value of durable property is false.

You might also think that setting durable to true on the exchanges and queues is all you need

to do to make your messages survive a reboot, but you’d be wrong. Whereas queues and exchanges

*must* be durable to allow messages to survive reboot, it isn’t enough on its own.

A message that can survive a crash of the AMQP broker is called *persistent*. You flag a

message as persistent by setting the *delivery mode* option of the message to 2 (your

AMQP client may use a human-friendly constant instead) before publishing it. At this

point, the message is indicated as persistent, but it must be published to an exchange

that’s durable and arrive in a queue that’s durable to survive.

Rabbit to survive a crash, the message must

 Have its delivery mode option set to 2 (persistent)

 Be published into a durable exchange

 Arrive in a durable queue

You might be thinking that you should use persistent messaging for all of your messages.

You could do that, but you’d pay a price for ensuring your messages survive

Rabbit restarts: performance. The act of writing messages to disk is much slower than

just storing them in RAM, and will significantly decrease the number of messages per

second your RabbitMQ server can process. It’s not uncommon to see a 10x or more

decrease in message throughput when using persistency.

*Transaction*:

A concept that’s related to the durability of a message is the AMQP *transaction*. So

far we’ve talked about marking messages, queues, and exchanges as durable. That’s all

well and good for keeping a message safe once RabbitMQ has it in its custody, but

since a publish operation returns no response to the producer, how do you know if

the broker has persisted the durable message to disk? Should the broker die before it

can write the message to disk, the message would be lost and you wouldn’t know.

That’s where transactions come in. When you absolutely need to be sure the

broker has the message in custody(and has routed the message to all matching subscribed queues)

before you move on to another task, you need to wrap it in a transaction. If you come from a

database background, it’s important not to confuse AMQP transactions

with what “transaction” means in most databases. In AMQP, after you place a

channel into transaction mode, you send it the publish you want to confirm, followed

by zero or more other AMQP commands that should be executed or ignored depending

on whether the initial publish succeeded. Once you’ve sent all of the commands,

you *commit* the transaction. If the transaction’s initial publish succeeds, then the channel

will complete the other AMQP commands in the transaction. If the publish fails,

none of the other AMQP commands will be executed. Transactions close the “last

mile” gap between producers publishing messages and RabbitMQ committing them

to disk, but there’s a better way to close that gap.

*publisher confirm:*

Not only can using transactions drop your message throughput by a factor of 2–10x, but they

Also make your producer app synchronous, which is one of the things you’re trying to get

rid of with messaging.

Knowing all of this, the guys at RabbitMQ decided to come up

with a better way to ensure message delivery: *publisher confirms*.2 Similar to transactions,

you have to tell Rabbit to place the channel into confirm mode, and you can’t turn it

off without re-creating the channel. Once a channel is in confirm mode, every message

published on the channel will be assigned a unique ID number (starting at 1).

Once the message has been delivered to all queues that have bindings matching the

message’s routing key, the channel will issue a publisher confirm to the producer app

(containing the message’s unique ID). This lets the producer know the message has

been safely queued at all of its destinations. If the message and the queues are durable,

the confirm is issued only after the queues have written the message to disk. The

major benefit of publisher confirms is that they’re asynchronous. Once a message has

been published, the producer app can go on to the next message while waiting for the

confirm. When the confirm for that message is finally received, a callback function in

the producer app will be fired so it can wake up and handle the confirmation. If an

internal error occurs inside Rabbit that causes a message to be lost, Rabbit will send a

message *nack* (not acknowledged) that’s like a publisher confirm (it has the message’s

unique ID) but indicates the message was lost. Also, since there’s no concept of message

rollback (as with transactions), publisher confirms are much lighter weight and

have an almost negligible performance hit on the Rabbit broker.

**Installing the management plugin**

$ sudo rabbitmq-plugins enable rabbitmq\_management

Then restart rabbitmq server

http://<hostname>:15672

**Consumer Prefetch**

A more natural and efficient way to limit unacknowledged messages.

AMQP specifies the basic.qos method to allow you to limit the number of unacknowledged messages on a channel (or connection) when consuming (aka "prefetch count").

### Example - single consumer

The following basic (Java) example will receive a maximum of 10 unacknowledged messages at once:

Channel channel = ...;

Consumer consumer = ...;

channel.basicQos(10); // Per consumer limit

channel.basicConsume("my-queue", false, consumer);

### Example - multiple independent consumers

This example starts two consumers on the same channel, each of which will independently receive a maximum of 10 unacknowledged messages at once:

Channel channel = ...;

Consumer consumer1 = ...;

Consumer consumer2 = ...;

channel.basicQos(10); // Per consumer limit

channel.basicConsume("my-queue1", false, consumer1);

channel.basicConsume("my-queue2", false, consumer2);

### Example - multiple consumers sharing the limit

The AMQP specification does not explain what happens if you invoke basic.qos multiple times with differentglobal values. RabbitMQ interprets this as meaning that the two prefetch limits should be enforced independently of each other; consumers will only receive new messages when neither limit on unacknowledged messages has been reached.

For example:

Channel channel = ...;

Consumer consumer1 = ...;

Consumer consumer2 = ...;

channel.basicQos(10, false); // Per consumer limit

channel.basicQos(15, true); // Per channel limit

channel.basicConsume("my-queue1", false, consumer1);

channel.basicConsume("my-queue2", false, consumer2);

These two consumers will only ever have 15 unacknowledged messages between them, with a maximum of 10 messages for each consumer. This will be slower than the above examples, due to the additional overhead of coordinating between the channel and the queues to enforce the global limit.

(Note that the default value for the global flag is false in most APIs. channel.basicQos(maxMessage,globalFlag))

# Consumer Priorities

Consumer priorities allow you to ensure that high priority consumers receive messages while they are active, with messages only going to lower priority consumers when the high priority consumers block.

Normally, active consumers connected to a queue receive messages from it in a round-robin fashion. When consumer priorities are in use, messages are delivered round-robin if multiple active consumers exist with the same high priority.

### Definition of active consumers

The above paragraphs refer to consumers as being *active* or *blocked*. At any moment, any given consumer is either one or the other. An active consumer is one which could receive a message without waiting. A consumer becomes blocked if it cannot receive messages - because its channel has reached the maximum number of unacknowledged messages after issuing basic.qos, or simply because of network congestion.

Therefore for each queue, at least one of three things must be true:

* There are no active consumers
* The queue is empty
* The queue is busy delivering messages to consumers

Note that consumers can switch between active and blocked many times per second. We therefore don't expose whether a consumer is active or blocked through the management plugin or rabbitmqctl.

When consumer priorities are in use, you can expect your highest priority consumers to receive all the messages until they become blocked, at which point lower priority consumers will start to receive some. It's important to understand that RabbitMQ will still prioritise delivering messages - it will not wait for a high priority blocked consumer to become unblocked if there is an active lower priority consumer ready.

### Using consumer priorities

Set the x-priority argument in the basic.consume method to an integer value. Consumers which do not specify a value have priority 0. Larger numbers indicate higher priority, and both positive and negative numbers can be used.

For example (in Java):

Channel channel = ...;

Consumer consumer = ...;

Map<String, Object> args = new HashMap<String, Object>();

args.put("x-priority", 10);

channel.basicConsume("my-queue", false, args, consumer);

This creates a new consumer with priority 10.

**delivery-tag:**

It is like message id for a particular channel

The server-assigned and channel-specific delivery tag

* The delivery tag is valid only within the channel from which the message was received. I.e. a client MUST NOT receive a message on one channel and then acknowledge it on another.
* The server MUST NOT use a zero value for delivery tags. Zero is reserved for client use, meaning "all messages so far received".

# Negative Acknowledgements

The AMQP specification defines the basic.reject method that allows clients to reject individual, delivered messages, instructing the broker to either discard them or requeue them. Unfortunately, basic.rejectprovides no support for negatively acknowledging messages in bulk.

To solve this, RabbitMQ supports the basic.nack method that provides all the functionality of basic.rejectwhilst also allowing for bulk processing of messages.

To reject messages in bulk, clients set the multiple flag of the basic.nack method to true. The broker will then reject all unacknowledged, delivered messages up to and including the message specified in thedelivery\_tag field of the basic.nack method. In this respect, basic.nack complements the bulk acknowledgement semantics of basic.ack.

This example rejects a single message, asking the broker to requeue it:

GetResponse gr = channel.basicGet("some.queue", false);

channel.basicNack(gr.getEnvelope().getDeliveryTag(), false, true);

This example rejects two messages with a single call to the broker (the second argument on basicNack is themultiple flag):

GetResponse gr1 = channel.basicGet("some.queue", false);

GetResponse gr2 = channel.basicGet("some.queue", false);

channel.basicNack(gr2.getEnvelope().getDeliveryTag(), true, true);

Signature of ack methoods:

void **basicNack**(long deliveryTag,

boolean multiple,

boolean requeue)

throws java.io.IOException

void **basicAck**(long deliveryTag,

boolean multiple)

throws java.io.IOException

# Alternate Exchanges

It is sometimes desirable to let clients handle messages that an exchange was unable to route (i.e. either because there were no bound queues our no matching bindings). Typical examples of this are

* detecting when clients accidentally or maliciously publish messages that cannot be routed
* "or else" routing semantics where some messages are handled specially and the rest by a generic handler

RabbitMQ's Alternate Exchange ("AE") feature addresses these use cases.

For any given exchange, an AE can be defined by clients using the exchange's arguments, or in the server using [**policies**](https://www.rabbitmq.com/parameters.html#policies). In the case where both policy and arguments specify an AE, the one specified in arguments overrules the one specified in policy.

## Configuration using arguments

When creating an exchange the name of an AE can be optionally supplied in the exchange.declare method'sarguments table by specifying a key of 'alternate-exchange' and a value of type 'S' (string) containing the name.

When an AE has been specified, in addition to the usual configure permissions on the declared exchange, the user needs to have read permissions on that exchange and write permissions on the AE.

For example:

Map<String, Object> args = new HashMap<String, Object>();

args.put("alternate-exchange", "my-ae");

channel.exchangeDeclare("my-direct", "direct", false, false, args);

channel.exchangeDeclare("my-ae", "fanout");

channel.queueDeclare("routed");

channel.queueBind("routed", "my-direct", "key1");

channel.queueDeclare("unrouted");

channel.queueBind("unrouted", "my-ae", "");

In the above fragment of Java code we create a direct exchange 'my-direct' that is configured with an AE called 'my-ae'. The latter is declared as a fanout exchange. We bind one queue 'routed' to 'my-direct' with a binding key of 'key1', and a queue 'unrouted' to 'my-ae'.

# Sender-selected Distribution

The routing logic in AMQP does not offer a way for message publishers to select intended recipients. This is the equivalent of entering multiple recipients in the "CC" field of an email. The RabbitMQ broker treats two headers in a special way to overcome this limitation.

The values associated with the "CC" and "BCC" header keys will be added to the routing key if they are present. The message will be routed to all destinations matching the routing key supplied as a parameter to the basic.publish method, as well as the routes supplied in the "CC" and "BCC" headers. The type of "CC" and "BCC" values must be an array of **[longstr](https://www.rabbitmq.com/amqp-0-9-1-reference.html" \l "domain.longstr)** and these keys are case-sensitive. If the header does not contain "CC" or "BCC" keys then this extension has no effect.

The "BCC" key and value will be removed from the message prior to delivery, offering some confidentiality among consumers. This feature is a violation of the AMQP specification which forbids any message modification, including headers. This feature imposes a small performance penalty.

This extension is independent of the client. Any AMQP client with the ability to interoperate with RabbitMQ and set header values can make use of this extension.

Java sample code:

1: BasicProperties props = new BasicProperties();

2: Map<String, Object> headers = new HashMap<String, Object>();

3: List<String> ccList = new ArrayList<String>();

4: ccList.add("routingkey2");

5: ccList.add("routingkey3");

6: headers.put("CC", ccList);

7: props.setHeaders(headers);

8: channel.basicPublish(exchange, "routingkey1", props, payload);

# Time-To-Live Extensions

RabbitMQ allows you to set Time To Live for both messages and queues.

## Per-Queue Message TTL

TTL can be set for a given queue by setting the x-message-ttl argument to queue.declare, or by setting themessage-ttl [**policy**](https://www.rabbitmq.com/parameters.html#policies). A message that has been in the queue for longer than the configured TTL is said to bedead. Note that a message routed to multiple queues can die at different times, or not at all, in each queue in which it resides. The death of a message in one queue has no impact on the life of the same message in other queues.

The server guarantees that dead messages will not be included in any basic.get-ok or basic.delivermethods. Further, the server will try to reap messages at or shortly after their TTL-based expiry.

The value of the TTL argument or policy must be a non-negative integer (0 <= n), describing the TTL period in milliseconds. Thus a value of 1000 means that a message added to the queue will live in the queue for 1 second or until it is delivered to a consumer. The argument can be of AMQP type short-short-int, short-int, long-int, or long-long-int.

This example in Java creates a queue in which messages may reside for at most 60 seconds:

Map<String, Object> args = new HashMap<String, Object>();

args.put("x-message-ttl", 60000);

channel.queueDeclare("myqueue", false, false, false, args);

To specify a TTL using policy, add the key "message-ttl" to a policy definition. For example:

|  |  |
| --- | --- |
| **rabbitmqctl** | rabbitmqctl set\_policy TTL ".\*" '{"message-ttl":60000}' --apply-to queues |
| **rabbitmqctl (Windows)** | rabbitmqctl set\_policy TTL ".\*" "{""message-ttl"":60000}" --apply-to queues |

This applies a TTL of 60 seconds to all queues.

The original expiry time of a message is preserved if it is requeued (for example due to the use of an AMQP method that features a requeue parameter, or due to a channel closure).

Setting the TTL to 0 causes messages to be expired upon reaching a queue unless they can be delivered to a consumer immediately. Thus this provides an alternative to basic.publish's immediate flag, which the RabbitMQ server does not support. Unlike that flag, no basic.returns are issued, and if a dead letter exchange is set then messages will be dead-lettered.

## Per-Message TTL

A TTL can be specified on a per-message basis, by setting the expiration field in the [**basic**](https://www.rabbitmq.com/amqp-0-9-1-reference.html#class.basic) AMQP class when sending a **[basic.publish](https://www.rabbitmq.com/amqp-0-9-1-reference.html" \l "basic.publish)**.

The value of the expiration field describes the TTL period in milliseconds. The same constraints as forx-message-ttl apply. Since the expiration field must be a string, the broker will (only) accept the string representation of the number.

When both a per-queue and a per-message TTL are specified, the lower value between the two will be chosen.

This example in Java publishes a message which can reside in the queue for at most 60 seconds:

byte[] messageBodyBytes = "Hello, world!".getBytes();

AMQP.BasicProperties properties = new AMQP.BasicProperties();

properties.setExpiration("60000");

channel.basicPublish("my-exchange", "routing-key", properties, messageBodyBytes);

### Caveats

While consumers never see expired messages, only when expired messages reach the head of a queue will they actually be discarded (or dead-lettered). When setting a per-queue TTL this is not a problem, since expired messages are always at the head of the queue. When setting per-message TTL however, expired messages can queue up behind non-expired ones until the latter are consumed or expired. Hence resources used by such expired messages will not be freed, and they will be counted in queue statistics (e.g. the number of messages in the queue).

## Queue TTL

Expiry time can be set for a given queue by setting the x-expires argument to queue.declare, or by setting the expires [**policy**](https://www.rabbitmq.com/parameters.html#policies). This controls for how long a queue can be unused before it is automatically deleted.Unused means the queue has no consumers, the queue has not been redeclared, and basic.get has not been invoked for a duration of at least the expiration period. This can be used, for example, for RPC-style reply queues, where many queues can be created which may never be drained.

The server guarantees that the queue will be deleted, if unused for at least the expiration period. No guarantee is given as to how promptly the queue will be removed after the expiration period has elapsed. Leases of durable queues restart when the server restarts.

The value of the x-expires argument or expires policy describes the expiration period in milliseconds. It must be a positive integer (unlike message TTL it cannot be 0). Thus a value of 1000 means a queue which is unused for 1 second will be deleted.

This example in Java creates a queue which expires after it has been unused for 30 minutes.

Map<String, Object> args = new HashMap<String, Object>();

args.put("x-expires", 1800000);

channel.queueDeclare("myqueue", false, false, false, args);

The following policy does the same thing for all queues:

|  |  |
| --- | --- |
| **rabbitmqctl** | rabbitmqctl set\_policy expiry ".\*" '{"expires":1800000}' --apply-to queues |
| **rabbitmqctl (Windows)** | rabbitmqctl set\_policy expiry ".\*" "{""expires"":1800000}" --apply-to queues |

# Dead Letter Exchanges

Messages from a queue can be 'dead-lettered'; that is, republished to another exchange when any of the following events occur:

* The message is rejected (basic.reject or basic.nack) with requeue=false,
* The TTL for the message expires; or
* The queue length limit is exceeded.

Dead letter exchanges (DLXs) are normal exchanges. They can be any of the usual types and are declared as usual.

For any given queue, a DLX can be defined by clients using the queue's arguments, or in the server using[**policies**](https://www.rabbitmq.com/parameters.html#policies). In the case where both policy and arguments specify a DLX, the one specified in arguments overrules the one specified in policy.

## Configuration using arguments

To set the dead letter exchange for a queue, set the x-dead-letter-exchange argument to the name of the exchange:

channel.exchangeDeclare("some.exchange.name", "direct");

Map<String, Object> args = new HashMap<String, Object>();

args.put("x-dead-letter-exchange", "some.exchange.name");

channel.queueDeclare("myqueue", false, false, false, args);

The code above declares a new exchange called some.exchange.name and sets this new exchange as the dead letter exchange for a newly created queue. Note that the exchange does not have to be declared when the queue is declared, but it should exist by the time messages need to be dead-lettered; if it is missing then, the messages will be silently dropped.

You may also specify a routing key to be used when dead-lettering messages. If this is not set, the message's own routing keys will be used.

args.put("x-dead-letter-routing-key", "some-routing-key");

When a dead letter exchange has been specified, in addition to the usual configure permissions on the declared queue, the user needs to have read permissions on that queue and write permissions on the dead letter exchange. Permissions are verified at the time of queue declaration.

## Configuration using policy

To specify a DLX using policy, add the key "dead-letter-exchange" to a policy definition. For example:

|  |  |
| --- | --- |
| **rabbitmqctl** | rabbitmqctl set\_policy DLX ".\*" '{"dead-letter-exchange":"my-dlx"}' --apply-to queues |
| **rabbitmqctl (Windows)** | rabbitmqctl set\_policy DLX ".\*" "{""dead-letter-exchange"":""my-dlx""}" --apply-to queues |

This applies the DLX "my-dlx" to all queues.

Similarly, an explicit routing key can be specified by adding the key "dead-letter-routing-key" to the policy.

Policies can also be defined using the management plugin, see the [**policy documentation**](https://www.rabbitmq.com/parameters.html#policies) for more details.

## Routing Dead-Lettered Messages

Dead-lettered messages are routed to their dead letter exchange either:

* with the routing key specified for the queue they were on; or, if this was not set,
* with the same routing keys they were originally published with.

For example, if you publish a message to an exchange with routing key foo, and that message is dead-lettered, it will be published to its dead letter exchange with routing key foo. If the queue the message originally landed on had been declared with x-dead-letter-routing-key set to bar, then the message will be published to its dead letter exchange with routing key bar.

Note that, if a specific routing key was not set for the queue, messages on it are dead-lettered with all their original routing keys. This includes routing keys added by the CC and BCC headers (see [**Sender-selected distribution**](https://www.rabbitmq.com/sender-selected.html) for details on these two headers).

Dead-lettered messages are re-published with publisher confirms turned on internally so, the dead-letter queues the messages eventually land on must confirm the messages before they are removed from the original queue. In other words, the publishing queue will not remove messages before the dead-letter queues acknowledge receiving them (see [**Confirms**](https://www.rabbitmq.com/confirms.html) for details on the guarantees made). Note that, in the event of an unclean broker shutdown, the same message may be duplicated on both the original queue and on the dead-letter queues.

It is possible to form a cycle of dead-letter queues. For instance, this can happen when a queue dead-letters messages to the default exchange without specifiying a dead-letter routing key. Messages in such cycles (i.e. messages that reach the same queue twice) will be dropped if the entire cycle is due to message expiry.

## Dead-Lettered Messages

Dead-lettering a message modifies its headers:

* the exchange name is replaced with that of the latest dead-letter exchange,
* the routing key may be replaced with that specified in a dead letter queue,
* if the above happens, the CC header will also be removed, and
* the BCC header will be removed as per [**Sender-selected distribution**](https://www.rabbitmq.com/sender-selected.html).

The dead-lettering process adds an array to the header of each dead-lettered message named x-death. This array contains an entry for each time the message was dead-lettered. Each such entry is a table that consists of several fields:

* queue - the name of the queue the message was in before it was dead-lettered,
* reason - see below,
* time - the date and time the message was dead lettered as a 64-bit AMQP format timestamp,
* exchange - the exchange the message was published to (note that this will be a dead letter exchange if the message is dead lettered multiple times), and
* routing-keys - the routing keys (including CC keys but excluding BCC ones) the message was published with.
* original-expiration (if the message was dead-letterered due to [**per-message TTL**](https://www.rabbitmq.com/ttl.html#per-message-ttl)) - the originalexpiration property of the message. The expiration property is removed from the message on dead-lettering in order to prevent it from expiring again in any queues it is routed to.

The reason is a name describing why the message was dead-lettered and is one of the following:

* rejected - the message was rejected with requeue=false,
* expired - the TTL of the message expired; or
* maxlen - the maximum allowed queue length was exceeded.

Note that the array is sorted most-recent-first, so the most recent dead-lettering will be recorded in the first entry.

# Queue Length Limit

The maximum length of a queue can be limited to a set number of messages, or a set number of bytes (the total of all message body lengths, ignoring message properties and any overheads), or both.

For any given queue, the maximum length (of either type) can be defined by clients using the queue's arguments, or in the server using [**policies**](https://www.rabbitmq.com/parameters.html#policies). In the case where both policy and arguments specify a maximum length, the minimum of the two values is applied.

In all cases the number of *ready* messages is used; unacknowledged messages do not count towards the limit. The fields messages\_ready and message\_bytes\_ready from rabbitmqctl list\_queues and the management API show the values that would be limited.

Messages will be dropped or dead-lettered from the front of the queue to make room for new messages once the limit is reached.

### Configuration using arguments

Maximum number of messages can be set by supplying the x-max-length queue declaration argument with a non-negative integer value.

Maximum length in bytes can be set by supplying the x-max-length-bytes queue declaration argument with a non-negative integer value.

If both arguments are set then both will apply; whichever limit is hit first will be enforced.

This example in Java declares a queue with a maximum length of 10 messages:

Map<String, Object> args = new HashMap<String, Object>();

args.put("x-max-length", 10);

channel.queueDeclare("myqueue", false, false, false, args);

### Configuration using policy

To specify a maximum length using policy, add the key max-length and / or max-length-bytes to a policy definition. For example:

|  |  |
| --- | --- |
| **rabbitmqctl** | rabbitmqctl set\_policy Ten "^one-meg$" '{"max-length-bytes":1000000}' --apply-to queues |
| **rabbitmqctl (Windows)** | rabbitmqctl set\_policy Ten "^one-meg$" "{""max-length-bytes"":1000000}" --apply-to queues |

This ensures the queue called one-meg can contain no more than 1MB of message bodies.

Policies can also be defined using the management plugin, see the [**policy documentation**](https://www.rabbitmq.com/parameters.html#policies) for more details.

# Priority Queue Support

RabbitMQ has priority queue implementation in the core as of version 3.5.0.

You can declare priority queues using the x-max-priority argument. This argument should be an integer indicating the maximum priority the queue should support. For example, using the Java client:

Channel ch = ...;

Map<String, Object> args = new HashMap<String, Object>();

args.put("x-max-priority", 10);

ch.queueDeclare("my-priority-queue", true, false, false, args);

You can then publish prioritised messages using the priority field of basic.properties. Larger numbers indicate higher priority.

Because the on-disk format for a priority queue is different, priority queues can only be defined by arguments, not policies. Queues can never change the number of priorities they support.

## Behaviour

The AMQP 0-9-1 spec is a bit vague about how priorities work. It says that all queues MUST support at least 2 priorities, and MAY support up to 10. It does not define how messages without a priority property are treated.

In contrast to the AMQP 0-9-1 spec, RabbitMQ queues by default do not support priorities. When creating priority queues, you can specify as many priority levels as you like. Note that:

* There is some in-memory and on-disk cost per priority level per queue. There is also an additional CPU cost, especially when consuming, so you may not wish to create huge numbers of levels.
* The message priority field is defined as an unsigned byte, so in practice priorities should be between 0 and 255.

Messages without a priority property are treated as if their priority were 0. Messages with a priority which is higher than the queue's maximum are treated as if they were published with the maximum priority.

## Interaction with consumers

It's important to understand how consumers work when working with priority queues. By default, consumers may be sent a large number of messages before they acknowledge any, limited only by network backpressure.

So if such a hungry consumer connects to an empty queue to which messages are subsequently published, the messages may not spend any time at all waiting in the queue. In this case the priority queue will not get any opportunity to prioritise them.

In most cases you will want to use the basic.qos method in manual acknowledgement mode on your consumers, to limit the number of messages that can be out for delivery at any time and thus allow messages to be prioritised.

## Interaction with other features

In general priority queues have all the features of standard RabbitMQ queues: they support persistence, paging, mirroring, and so on. There are a couple of interactions that should be noted though:

* [**Messages which should expire**](https://www.rabbitmq.com/ttl.html) will still only expire from the head of the queue. This means that unlike with normal queues, even per-queue TTL can lead to expired lower-priority messages getting stuck behind non-expired higher priority ones. These messages will never be delivered, but they will appear in queue statistics.
* [**Queues which have a max-length set**](https://www.rabbitmq.com/maxlength.html) will, as usual, drop messages from the head of the queue to enforce the limit. This means that higher priority messages might be dropped to make way for lower priority ones, which might not be what you would expect.

A message from the queue will be delivered to one consumer only. Ie: once the message makes its way to the queue - it won't be copied (broadcasted) to multiple consumers.

If you want to do broadcast - you have to use multiple queues.

### **Passive or active queues?**

In the previous RabbitMQ examples I have written, I have always declared queues calling Channel.queueDeclare(). But it is also possible to do it calling Channel.queueDeclarePassive(). As you could easily guess, the first method is for an active queue declaration, and the second for a passive one. The point of this post is: what a passive queue is and how to choose between declaring a queue as passive or active.  
  
If you play with queue setting, one day or another you will end up getting an exception, due to the fact that once a queue is created, you can't access it specifying a different setting:

java.io.IOException

at com.rabbitmq.client.impl.AMQChannel.wrap(AMQChannel.java:106)

at com.rabbitmq.client.impl.AMQChannel.wrap(AMQChannel.java:102)

at com.rabbitmq.client.impl.AMQChannel.exnWrappingRpc(AMQChannel.java:124)

...

Caused by: com.rabbitmq.client.ShutdownSignalException: channel error;

reason: {#method<channel.close>(reply-code=406, reply-text=PRECONDITION\_FAILED -

parameters for queue 'ack' in vhost '/' not equivalent, class-id=50, method-id=10),

null, "[B@19d3b3a"}

at com.rabbitmq.utility.ValueOrException.getValue(ValueOrException.java:67)

...

In this case I tried to access a queue called "ack", declared in the default virtual host, passing the parameter "autodelete" set to true, where it was created with false.  
  
When this happens, you could have a look to the RabbitMQ administrator, specifically in the Queues tab, to get all the available information on your queue.  
  
But back to the main theme of the post.  
  
If a queue is declared passively, RabbitMQ assumes it already exists, and we just want to establish a connection to it. What if you call Channel.queueDeclarePassive() and the queue is not there? I guess you already know the answer, you get a fat exception:

java.io.IOException

at com.rabbitmq.client.impl.AMQChannel.wrap(AMQChannel.java:106)

at com.rabbitmq.client.impl.AMQChannel.wrap(AMQChannel.java:102)

at com.rabbitmq.client.impl.AMQChannel.exnWrappingRpc(AMQChannel.java:124)

...

Caused by: com.rabbitmq.client.ShutdownSignalException: channel error;

reason: {#method<channel.close>(reply-code=404, reply-text=NOT\_FOUND -

no queue 'ack' in vhost '/', class-id=50, method-id=10), null, "[B@140fee"}

at com.rabbitmq.utility.ValueOrException.getValue(ValueOrException.java:67)

...

Close to the exception shown above, but here the reply text is a NOT\_FOUND that shows clearly what the problem is. Rabbit was running happily assuming the queue was there, but actually it wasn't.  
  
So, we use a passive queue declare when we can safely assume the queue is already there, and its non-existence would considered an exceptional, almost catastrophic, event.  
  
The safer active queue declaration has the obvious downside of being more expensive, and requiring the user to provide the setting each time a queue declaration is issued. An hybrid approach (active on the server, passive on the client) could make sense if the application has a rigid protocol ensuring that a component would always start before the other(s).