Consumers

Overview

This guide covers various topics related to consumers:

and more.

Terminology

The term "consumer" means different things in different contexts. In general, in the context of messaging and streaming, a consumer is an application (or application instance) that consumes and acknowledges messages. The same application can also publish messages and thus be a publisher at the same time.

Messaging protocols also have the concept of a lasting subscription for message delivery. Subscription is one term commonly used to describe such entity. Consumer is another. Messaging protocols supported by RabbitMQ use both terms but RabbitMQ documentation tends to prefer the latter.

In this sense a consumer is a subscription for message delivery that has to be registered before deliveries begin and can be cancelled by the application.

The Basics

RabbitMQ is a messaging broker. It accepts messages from publishers, routes them and, if there were queues to route to, stores them for consumption or immediately delivers to consumers, if any.

Consumers consume from queues. In order to consume messages there has to be a queue. When a new consumer is added, assuming there are already messages ready in the queue, deliveries will start immediately.

The target queue can be empty at the time of consumer registration. In that case first deliveries will happen when new messages are enqueued.

An attempt to consume from a non-existent queue will result in a channel-level exception with the code of `404 Not Found` and render the channel it was attempted on to be closed.

Consumer Tags

Every consumer has an identifier that is used by client libraries to determine what handler to invoke for a given delivery. Their names vary from protocol to protocol. Consumer tags and subscription IDs are two most commonly used terms. RabbitMQ documentation tends to use the former.

Consumer tags are also used to cancel consumers.

Consumer Lifecycle

Consumers are meant to be long lived: that is, throughout the lifetime of a consumer it receives multiple deliveries. Registering a consumer to consume a single message is not optimal.

Consumers are typically registered during application startup. They often would live as long as their connection or even application runs.

Consumers can be more dynamic and register in reaction to a system event, unsubscribing when they are no longer necessary. This is common with WebSocket clients used via Web STOMP and Web MQTT plugins, mobile clients and so on.

Connection Recovery

Client can lose their connection to RabbitMQ. When connection loss is detected, message delivery stops.

Some client libraries offer automatic connection recovery features that involves consumer recovery. Java, .NET and Bunny are examples of such libraries. While connection recovery cannot cover 100% of scenarios and workloads, it generally works very well for consuming applications and is recommended.

With other client libraries application developers are responsible for performing connection recovery. Usually the following recovery sequence works well:

- Recover connection
- Recover channels
- Recover queues
- Recover exchanges
- Recover bindings
- Recover consumers

In other words, consumers are usually recovered last, after their target queues and those queues' bindings are in place.

Registering a Consumer (Subscribing, "Push API")

Applications can subscribe to have RabbitMQ push enqueued messages (deliveries) to them. This is done by registering a consumer (subscription) on a queue. After a subscription is in place, RabbitMQ will begin delivering messages. For each delivery a user-provided handler will be invoked. Depending on the client library used this can be a user-provided function or object that adheres to a certain interface.

A successful subscription operation returns a subscription identifier (consumer tag). It can later be used to cancel the consumer.

Java Client

See Java client guide for examples.

.NET Client

See .NET client guide for examples.

Message Properties and Delivery Metadata

Every delivery combines message metadata and delivery information. Different client libraries use slightly different ways of providing access to those properties. Typically delivery handlers have access to a delivery data structure.

The following properties are delivery and routing details; they are not message properties per se and set by RabbitMQ at routing and delivery time:

Property	Type	Description
Delivery tag	Positive integer	Delivery identifier, see Confirms.
Redelivered	Boolean	Set to `true` if this message was previously delivered and requeued
Exchange	String	Exchange which routed this message
Routing key	String	Routing key used by the publisher
Consumer tag	String	Consumer (subscription) identifier

The following are message properties. Most of them are optional. They are set by publishers at the time of publishing:

Property	Type	Description	Required?
Delivery mode	Enum (1 or 2)	2 for "persistent", 1 for "transient". Some client libraries expose this property as a boolean or enum.	Yes
Type	String	Application-specific message type, e.g. "orders.created"	No
Headers	Map (string => any)	An arbitrary map of headers with string header names	No
Content type	String	Content type, e.g. "application/json". Used by applications, not core RabbitMQ	No
Content encoding	String	Content encoding, e.g. "gzip". Used by applications, not core RabbitMQ	No
Message ID	String	Arbitrary message ID	No
Correlation ID	String	Helps correlate requests with responses, see tutorial 6	No
Reply To	String	Carries response queue name, see tutorial 6	No
Expiration	String	Per-message TTL	No
Timestamp	Timestamp	Application-provided timestamp	No
User ID	String	User ID, validated if set	No
App ID	String	Application name	No

Message Types

The type property on messages is an arbitrary string that helps applications communicate what kind of message that is. It is set by the publishers at the time of publishing. The value can be any domain-specific string that publishers and consumers agree on.

RabbitMQ does not validate or use this field, it exists for applications and plugins to use and interpret.

Message types in practice naturally fall into groups, a dot-separated naming convention is common (but not required by RabbitMQ or clients), e.g. `orders.created` or `logs.line` or `profiles.image.changed`.

If a consumer gets a delivery of a type it cannot handle, it is highly advised to log such events to make troubleshooting easier.

Content Type and Encoding

The content (MIME media) type and content encoding fields allow publishers communicate how message payload should be deserialized and decoded by consumers.

RabbitMQ does not validate or use these fields, it exists for applications and plugins to use and interpret.

For example, messages with JSON payload should use `application/json`. If the payload is compressed with the LZ77 (GZip) algorithm, its content encoding should be `gzip`.

Multiple encodings can be specified by separating them with commas.

Acknowledgement Modes

When registering a consumer applications can choose one of two delivery modes:

- Automatic (deliveries require no acknowledgement, a.k.a. "fire and forget")
- Manual (deliveries require client acknowledgement)

Consumer acknowledgements are a subject of a separate documentation guide, together with publisher confirms, a closely related concept for publishers.

Limiting Simultaneous Deliveries with Prefetch

With manual acknowledgement mode consumers have a way of limiting how many deliveries can be "in flight" (in transit over the network or delivered but unacknowledged). This can avoid consumer overload.

This feature, together with consumer acknowledgements are a subject of a separate documentation guide.

The Consumer Capacity Metric

RabbitMQ management UI as well as monitoring data endpoints such as that for Prometheus scraping display a metric called consumer capacity (previously consumer utilisation) for individual queues.

The metric is computed as a fraction of the time that the queue is able to immediately deliver messages to consumers. It helps the operator notice conditions where it **may** be worthwhile adding more consumers (application instances) to the queue.

If this number is less than 100%, the queue leader replica may be able to deliver messages faster if:

- There were more consumers or
- The consumers spent less time processing deliveries or
- The consumer channels used a higher prefetch value

Consumer capacity will be 0% for queues that have no consumers. For queues that have online consumers but no message flow, the value will be 100%: the idea is that any number of consumers can sustain this kind of delivery rate.

Note that consumer capacity is merely a hint. Consumer applications can and should collect more specific metrics about their operations to help with sizing and any possible capacity changes.

Cancelling a Consumer (Unsubscribing)

To cancel a consumer its identifier (consumer tag) must be known.

After a consumer is cancelled there will be no future deliveries dispatched to it. Note that there can still be "in flight" deliveries dispatched previously. Cancelling a consumer will not discard them.

Java Client

See Java client guide for examples.

.NET Client

See .NET client guide for examples.

Fetching Individual Messages ("Pull API")

With AMQP 0-9-1 it is possible to fetch messages one by one using the `basic.get` protocol method. Messages are fetched in the FIFO order. It is possible to use automatic or manual acknowledgements, just like with consumers (subscriptions).

Fetching messages one by one is **highly discouraged** as it is **very inefficient** compared to regular long-lived consumers. As with any polling-based algorithm, it will be **extremely wasteful** in systems where message publishing is sporadic and queues can stay empty for prolonged periods of time.

When in doubt, prefer using a regular long-lived consumer.

Java Client

See Java client guide for examples.

.NET Client

See .NET client guide for examples.

Delivery Acknowledgement Timeout

In all currently supported RabbitMQ versions, a timeout is enforced on consumer delivery acknowledgement. This helps detect buggy (stuck) consumers that never acknowledge deliveries. Such consumers can affect node's on disk data compaction and potentially drive nodes out of disk space.

If a consumer does not ack its delivery for more than the timeout value (30 minutes by default), its channel will be closed with a `PRECONDITION_FAILED` channel exception. The error will be logged by the node that the consumer was connected to. All outstanding deliveries on that channel, from all consumers, will be requeued.

The timeout value is configurable in rabbitmq.conf (in milliseconds):

```
# 30 minutes in milliseconds
consumer_timeout = 1800000
```

```
# one hour in milliseconds
consumer_timeout = 3600000
```

The timeout can be deactivated using `advanced.config`. This is **not recommended**:

```
%% advanced.config
[
  {rabbit, [
    {consumer_timeout, undefined}
  ]}
].
```

Instead of disabling the timeout entirely, consider using a high value (for example, a few hours).

Exclusivity

When registering a consumer with an AMQP 0-9-1 client, the `exclusive` flag can be set to true to request the consumer to be the only one on the target queue. The call succeeds only if there's no consumer already registered to the queue at that time. This allows to make sure only one consumer at a time consumes from the queue.

If the exclusive consumer is cancelled or dies, this is the application responsibility to register a new one to keep on consuming from the queue.

If exclusive consumption *and* consumption continuity are required, single active consumer may be more appropriate.

Single Active Consumer

Single active consumer allows to have only one consumer at a time consuming from a queue and to fail over to another registered consumer in case the active one is cancelled or dies. Consuming with only one consumer is useful when messages must be consumed and processed in the same order they arrive in the queue.

A typical sequence of events would be the following:

- A queue is declared and some consumers register to it at roughly the same time.
- The very first registered consumer become the *single active consumer*: messages are dispatched to it and the other consumers are ignored.
- The single active consumer is cancelled for some reason or simply dies. One of the registered consumer becomes the new single active consumer and messages are now dispatched to it. In other terms, the queue fails over automatically to another consumer.

Note that without the single active consumer feature enabled, messages would be dispatched to all consumers using round-robin.

Single active consumer can be enabled when declaring a queue, with the `x-single-active-consumer` argument set to `true`, e.g. with the Java client:

```
Channel ch = ...;
Map<String, Object> arguments = new HashMap<String, Object>();
arguments.put("x-single-active-consumer", true);
ch.queueDeclare("my-queue", false, false, false, arguments);
```

Compared to AMQP exclusive consumer, single active consumer puts less pressure on the application side to maintain consumption continuity. Consumers just need to be registered and failover is handled automatically, there's no need to detect the active consumer failure and to register a new consumer.

The management UI and the CLI can report which consumer is the current active one on a queue where the feature is enabled.

Please note the following about single active consumer:

- There's no guarantee on the selected active consumer, it is picked up randomly, even if consumer priorities are in use.
- Trying to register a consumer with the exclusive consume flag set to true will result in an error if single active consumer is enabled on the queue.
- Messages are always delivered to the active consumer, even if it is too busy at some point. This can happen when using manual acknowledgment and `basic.qos`, the consumer may be busy dealing with the maximum number of unacknowledged messages it requested with `basic.qos`. In this case, the other consumers are ignored and messages are enqueued.
- It is not possible to enable single active consumer with a policy. Here is the reason why. Policies in RabbitMQ are dynamic by nature, they can come and go, enabling and disabling the features they declare. Imagine suddenly disabling single active consumer on a queue: the broker would start sending messages to inactive consumers and messages would be processed in parallel, exactly the opposite of what single active consumer is trying to achieve. As the semantics of single active consumer do not play well with the dynamic nature of policies, this feature can be enabled only when declaring a queue, with queue arguments.

Consumer Activity

The management UI and the `list_consumers` CLI command report an `active` flag for consumers. The value of this flag depends on several parameters.

- for classic queues, the flag is always `true` when single active consumer is not enabled.
- for quorum queues and when single active consumer is not enabled, the flag is `true` by default and is set to `false` if the node the consumer is connected to is suspected to be down.
- if single active consumer is enabled, the flag is set to `true` only for the current single active consumer, other consumers on the queue are waiting to be promoted if the active one goes away, so their active is set to `false`.

Priority

Normally, active consumers connected to a queue receive messages from it in a round-robin fashion.

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 are blocked, e.g. by effective prefetch setting.

When consumer priorities are in use, messages are delivered round-robin if multiple active consumers exist with the same high priority.

Consumer priorities are covered in a separate guide.

Exception Handling

Consumers are expected to handle any exceptions that arise during handling of deliveries or any other consumer operations. Such exceptions should be logged, collected and ignored.

If a consumer cannot process deliveries due to a dependency not being available or similar reasons it should clearly log so and cancel itself until it is capable of processing deliveries again. This will make the consumer's unavailability visible to RabbitMQ and monitoring systems.

Concurrency Considerations

Consumer concurrency is primarily a matter of client library implementation details and application configuration. With most client libraries (e.g. Java, .NET, Go, Erlang) deliveries are dispatched to a thread pool (or similar) that handles all asynchronous consumer operations. The pool usually has controllable degree of concurrency.

Java and .NET clients guarantee that deliveries on a single channel will be dispatched in the same order there were received regardless of the degree of concurrency. Note that once dispatched, concurrent processing of deliveries will result in a natural race condition between the threads doing the processing.

Certain clients (e.g. Bunny) and frameworks might choose to limit consumer dispatch pool to a single thread (or similar) to avoid a natural race condition when deliveries are processed concurrently. Some applications depend on strictly sequential processing of deliveries and thus must use concurrency factor of one or handle synchronisation in their own code. Applications that can process deliveries concurrently can use the degree of concurrency up to the number of cores available to them.

Queue Parallelism Considerations

A single RabbitMQ queue is bounded to a single core. Use more than one queue to improve CPU utilisation on the nodes. Plugins such as sharding and consistent hash exchange can be helpful in increasing parallelism.

Getting Help and Providing Feedback

If you have questions about the contents of this guide or any other topic related to RabbitMQ, don't hesitate to ask them on the RabbitMQ mailing list.

Help Us Improve the Docs <3

If you'd like to contribute an improvement to the site, its source is available on GitHub. Simply fork the repository and submit a pull request. Thank you!