Reliability Guide

This page explains how to use the various features of AMQP and RabbitMQ to achieve *reliable delivery* - to ensure that messages are always delivered, even encountering failure in any part of your system.

What Can Fail?

Network problems are probably the most common class of failure. Not only can networks fail, firewalls can interrupt idle connections, and network failures are not always detected immediately.

In addition to connectivity failures, the broker and client applications can experience hardware failure (or software can crash) at any time. Additionally, even if client applications keep running, logic errors can cause channel or connection errors which force the client to establish a new channel or connection and recover from the problem.

Connection Failures

In the event of a connection failure, the client will need to establish a new connection to the broker. Any channels opened on the previous connection will have been automatically closed and these will need re-opening too.

In general when connections fail, the client will be informed by the connection throwing an exception (or similar language construct). The official Java and .NET clients additionally provide callback methods to let you hear about connection failures in other contexts - Java provides the `ShutdownListener` callback on both `Connection` and `Channel` classes, and .NET client provides `IConnection.ConnectionShutdown` and `IModel.ModelShutdown` events for the same purpose.

Acknowledgements and Confirms

When a connection fails, messages may be in transit between client and server - they may be in the middle of being parsed or generated, in OS buffers, or on the wire. Messages in transit will be lost - they will need to be retransmitted. *Acknowledgements* let the server and clients know when to do this.

Acknowledgements can be used in both directions - to allow a consumer to indicate to the server that it has received / processed a message and to allow the server to indicate the same thing to the producer. RabbitMQ refers to the latter case as a "confirm".

Of course, TCP ensures that packets have been received, and will retransmit until they are - but that's just the network layer. Acknowledgements and confirms indicate that messages have been received *and acted upon*. An acknowledgement signals both the receipt of a message, and a transfer of ownership where the receiver assumes full responsibility for it.

Acknowledgements therefore have semantics - a consuming application should not acknowledge messages until it has done whatever it needs to do with them - recorded them in a database, forwarded them on, printed them onto paper or anything else. Once it does so, the broker is free to forget about the message.

Similarly, the broker will confirm messages once it has taken responsibility for them (see **here** for what that means).

Use of acknowledgements guarantees *at-least-once* delivery. Without acknowledgements, message loss is possible during publish and consume operations and only *at-most-once* delivery is guaranteed.

Heartbeats

In some types of network failure, packet loss can mean that disrupted TCP connections take some time to be detected by the operating system. AMQP offers a *heartbeat* feature to ensure that the application layer promptly finds out about disrupted connections (and also completely unresponsive peers). Heartbeats also defend against certain network equipment which may terminate "idle" TCP connections. In RabbitMQ versions 3.0 and higher, the broker will attempt to negotiate heartbeats by default (although the client can still veto them). Using earlier versions the client must be configured to request heartbeats.

At the Broker

In order to avoid losing messages in the broker we need to cope with broker restarts, broker hardware failure and *in extremis* even broker crashes.

To ensure that messages and broker definitions survive restarts, we need to ensure that they are on disk. The AMQP standard has a concept of durability for exchanges, queues and of persistent messages, requiring that a durable object or persistent message will survive a restart. More details about specific flags pertaining to durability and persistence can be found in the **AMQP Concepts Guide**.

Clustering and High Availability

If we need to ensure that our broker survives hardware failure, we can use RabbitMQ's clustering. In a RabbitMQ cluster, all definitions (of exchanges, bindings, users, etc) are mirrored across the entire cluster. Queues behave differently, by default residing only on a single node, but optionally being mirrored across several or all nodes. Queues remain visible and reachable from all nodes regardless of where they are located.

Mirrored queues replicate their contents across all configured cluster nodes, tolerating node failures seamlessly and without message loss (although see **this note on unsynchronised slaves**). However, consuming applications need to be aware that

when queues fail their consumers will be cancelled and they will need to reconsume - see **the documentation** for more details.

At the Producer

When using confirms, producers recovering from a channel or connection failure should retransmit any messages for which an acknowledgement has not been received from the broker. There is a possibility of message duplication here, because the broker might have sent a confirmation that never reached the producer (due to network failures, etc). Therefore consumer applications will need to perform deduplication or handle incoming messages in an idempotent manner.

Ensuring Messages are Routed

In some circumstances it can be important for producers to ensure that their messages are being routed to queues (although not always - in the case of a pub-sub system producers will just publish and if no consumers are interested it is correct for messages to be dropped).

To ensure messages are routed to a single known queue, the producer can just declare a destination queue and publish directly to it. If messages may be routed in more complex ways but the producer still needs to know if they reached at least one queue, it can set the `mandatory` flag on a `basic.publish`, ensuring that a `basic.return` (containing a reply code and some textual explanation) will be sent back to the client if no queues were appropriately bound.

Producers should also be aware that when publishing to a clustered node, if one or more destination queues that are bound to the exchange have mirrors in the cluster, it's possible to incur delays in the face of network failures between nodes, due to flow control between replicas and the master queue process. See **here** for more details.

At the Consumer

In the event of network failure (or a node crashing), messages can be duplicated, and consumers must be prepared to handle them. If possible, the simplest way to handle this is to ensure that your consumers handle messages in an idempotent way rather than explicitly deal with deduplication.

If a message is delivered to a consumer and then requeued (because it was not acknowledged before the consumer connection dropped, for example) then RabbitMQ will set the `redelivered` flag on it when it is delivered again (whether to the same consumer or a different one). This is a hint that a consumer *may* have seen this message before (although that's not guaranteed, the message may have made it out of the broker but not into a consumer before the connection dropped). Conversely if the `redelivered` flag is not set then it is guaranteed that the message has not been seen before. Therefore if a consumer finds it more expensive to deduplicate messages or process them in an idempotent manner, it can do this only for messages with the `redelivered` flag set.

Consumer Cancel Notification

Under some circumstances the server needs to be able to cancel a consumer - since the queue it was consuming from has been deleted, or has **failed over**. In this case the consumer should consume again but be aware that it may see messages again which it has already seen.

Note that consumer cancel notification is a RabbitMQ extension to AMQP, and as such may not be supported by all clients.

Messages That Cannot Be Processed

If a consumer determines that it cannot handle a message then it can *reject* it using `basic.reject` (or `basic.nack`), either asking the server to requeue it, or not (in which case the server might be configured to **dead-letter** it instead.

Distributed RabbitMQ

Rabbit provides two plugins to assist with distributing nodes over unreliable networks: **federation** and the **shovel**. Both are implemented as AMQP clients, so if you configure them to use confirms and acknowledgements, they will retransmit when necessary. Both will use confirms and acknowledgements by default.

When connecting clusters with federation or the shovel, it is desirable to ensure that the federation links and shovels tolerate node failures. Federation will automatically distribute links across the downstream cluster and fail them over on failure of a downstream node. In order to connect to a new upstream when an upstream node fails you can specify multiple redundant URIs for an upstream, or connect via a TCP load balancer.

When using the shovel, it is possible to specify redundant brokers in a source or destination clause; however it is not currently possible to make the shovel itself redundant. We hope to improve this situation in the future; in the mean time a new node can be brought up manually to run a shovel if the node it was originally running on fails.