Highly Available Queues

By default, queues within a RabbitMQ cluster are located on a single node (the node on which they were first declared). This is in contrast to exchanges and bindings, which can always be considered to be on all nodes. Queues can optionally be made *mirrored* across multiple nodes. Each mirrored queue consists of one *master* and one or more *slaves*, with the oldest slave being promoted to the new master if the old master disappears for any reason.

Messages published to the queue are replicated to all slaves. Consumers are connected to the master regardless of which node they connect to, with slaves dropping messages that have been acknowledged at the master. Queue mirroring therefore enhances availability, but does not distribute load across nodes (all participating nodes each do all the work).

This solution requires a RabbitMQ cluster, which means that it will not cope seamlessly with network partitions within the cluster and, for that reason, is not recommended for use across a WAN (though of course, clients can still connect from as near and as far as needed).

Configuring Mirroring

Queues have mirroring enabled via **policy**. Policies can change at any time; it is valid to create a non-mirrored queue, and then make it mirrored at some later point (and vice versa). There is a difference between a non-mirrored queue and a mirrored queue which does not have any slaves - the former lacks the extra mirroring infrastructure and will run faster.

You should be aware of the behaviour of **adding mirrors to a queue**.

To cause queues to become mirrored, you should create a policy which matches them and sets policy keys `ha-mode` and (optionally) `ha-params`. The following table explains the options for these keys:

ha-mode	ha-params	Result
`all`	(absent)	Queue is mirrored across all nodes in the cluster. When a new node is added to the cluster, the queue will be mirrored to that node.
`exactly`	*count*	Queue is mirrored to *count* nodes in the cluster. If there are less than *count* nodes in the cluster, the queue is mirrored to all nodes. If there are more than *count* nodes in the cluster, and a node containing a mirror goes down, then a new mirror will **not** be created on another node. (This is to prevent queues migrating across a cluster as it is brought down.)
`nodes`	*node names*	Queue is mirrored to the nodes listed in *node names*. If any of those node names are not a part of the cluster, this does not constitute an error. If none of the nodes in the list are online at the time when the queue is declared then the queue will be created on the node that the declaring client is connected to.

Whenever the HA policy for a queue changes it will endeavour to keep its existing mirrors as far as this fits with the new policy.

"nodes" policy and migrating masters

Note that setting or modifying a "nodes" policy can cause the existing master to go away if it is not listed in the new policy. In order to prevent message loss, RabbitMQ will keep the existing master around until at least one other slave has synchronised (even if this is a long time). However, once synchronisation has occured things will proceed just as if the node had failed: consumers will be disconnected from the master and will need to reconnect.

For example, if a queue is on `[A B]` (with A the master), and you give it a `nodes` policy telling it to be on `[C D]`, it will initially end up on `[A C D]`. As soon as the queue synchronises on its new mirrors `[C D]`, the master on A will shut down.

Exclusive queues

Exclusive queues will be deleted when the connection that declared them is closed. For this reason, it is not useful for an exclusive queue to be mirrored (or durable for that matter) since when the node hosting it goes down, the connection will close and the queue will need to be deleted anyway.

For this reason, exclusive queues are never mirrored (even if they match a policy stating that they should be). They are also never durable (even if declared as such).

Some examples

Policy where queues whose names begin with "`ha.`" are mirrored to all nodes in the cluster:

rabbitmqctl	`rabbitmqctl set_policy ha-all "^ha\." '{"ha-mode":"all"}'`
rabbitmqctl (Windows)	`rabbitmqctl set_policy ha-all "^ha\." "{""ha-mode"":""all""}"`
HTTP API	`PUT /api/policies/%2f/ha-all {"pattern":"^ha\.", "definition":{"ha-mode":"all"}}`
Web UI	▹ Navigate to Admin > Policies > Add / update a policy.

	▹ Enter "ha-all" next to Name, "^ha\." next to Pattern, and "ha-mode" = "all" in the first line next to Policy. ▹ Click Add policy.

Policy where queues whose names begin with "`two.`" are mirrored to any two nodes in the cluster, with **automatic synchronisation**:

rabbitmqctl	`rabbitmqctl set_policy ha-two "^two\." \` `'{"ha-mode":"exactly","ha-params":2,"ha-sync-mode":"automatic"}'`
rabbitmqctl (Windows)	`rabbitmqctl set_policy ha-two "^two\." ^` `"{""ha-mode"":""exactly"",""ha-params"":2,"ha-sync-mode":"automatic"}"`
HTTP API	`PUT /api/policies/%2f/ha-two` `{"pattern":"^two\.", "definition":{"ha-mode":"exactly", "ha-params":2,"ha-sync-mode":"automatic"}}`
Web UI	▹ Navigate to Admin > Policies > Add / update a policy. ▹ Enter "ha-two" next to Name and "^two\." next to Pattern. ▹ Enter "ha-mode" = "exactly" in the first line next to Policy, then "ha-params" = 2 in the second line, then "ha-sync-mode" = "automatic" in the third, and set the type on the second line to "Number". ▹ Click Add policy.

Unsynchronised Slaves

A node may join a cluster at any time. Depending on the configuration of a queue, when a node joins a cluster, queues may add a slave on the new node. At this point, the new slave will be empty: it will not contain any existing contents of the queue. Such a slave will receive new messages published to the queue, and thus over time will accurately represent the tail of the mirrored queue. As messages are drained from the mirrored queue, the size of the head of the queue for which the new slave is missing messages, will shrink until eventually the slave's contents precisely match the master's contents. At this point, the slave can be considered fully synchronised, but it is important to note that this has occured because of actions of clients in terms of draining the pre-existing head of the queue.

Thus a newly added slave provides no additional form of redundancy or availability of the queue's contents that existed before the slave was added, unless the queue has been explicitly synchronised. Since the queue becomes unresponsive while explicit synchronisation is occurring, it is preferable to allow active queues from which messages are being drained to synchronise naturally, and only explicitly synchronise inactive queues.

Configuring explicit synchronisation

Explicit synchronisation can be triggered in two ways: manually or automatically. If a queue is set to automatically synchronise it will synchronise whenever a new slave joins - becoming unresponsive until it has done so.

Queues can be set to automatically synchronise by setting the `ha-sync-mode` policy key to `automatic`. `ha-sync-mode` can also be set to `manual`. If it is not set then `manual` is assumed.

You can determine which slaves are synchronised with the following rabbitmqctl invocation:

```
rabbitmqctl list_queues name slave_pids synchronised_slave_pids
```

You can manually synchronise a queue with:

```
rabbitmqctl sync_queue name
```

And you can cancel synchronisation with:

```
rabbitmqctl cancel_sync_queue name
```

These features are also available through the management plugin.

Stopping nodes and synchronisation

If you stop a RabbitMQ node which contains the master of a mirrored queue, some slave on some other node will be promoted to the master (assuming there is one). If you continue to stop nodes then you will reach a point where a mirrored queue has no more slaves: it exists only on one node, which is now its master. If the mirrored queue was declared *durable* then, if its last remaining node is shutdown, durable messages in the queue will survive the restart of that node. In general, as you restart other nodes, if they were previously part of a mirrored queue then they will rejoin the mirrored queue.

However, there is currently no way for a slave to know whether or not its queue contents have diverged from the master to which it is rejoining (this could happen during a network partition, for example). As such, when a slave rejoins a mirrored queue, it throws away any durable local contents it already has and starts empty. Its behaviour is at this point the same as if it were a **new node joining the cluster**.

Mirrored queue implementation and semantics

As discussed, for each mirrored queue there is one *master* and several *slaves*, each on a different node. The slaves apply the operations that occur to the master in exactly the same order as the master and thus maintain the same state. All actions other than publishes go only to the master, and the master then broadcasts the effect of the actions to the slaves. Thus clients consuming from a mirrored queue are in fact consuming from the master.

Should a slave fail, there is little to be done other than some bookkeeping: the master remains the master and no client need take any action or be informed of the failure. Note that slave failures may not be detected immediately and the interruption of the per-connection flow control mechanism can delay message publication. The details are described **here**.

If the master fails, then one of the slaves must be promoted. At this point, the following happens:

- A slave is promoted to become the new master. The slave chosen for promotion is the eldest slave. As such, it has the best chance of being synchronised with the master. However, note that should there be no slave that is **synchronised** with the master, messages that only the master held will be lost.
- The slave considers all previous consumers to have been abruptly disconnected. As such, it requeues all messages that have been delivered to clients but are pending acknowledgement. This can include messages for which a client has issued acknowledgements: either the acknowledgement was lost on the wire before reaching the master, or it was lost during broadcast from the master to the slaves. In either case, the new master has no choice but to requeue all messages it thinks have not been acknowledged.
- Clients that have requested it are cancelled (see **below**).
- *As a result of the requeuing, clients that re-consume from the queue must be aware that they are likely to subsequently receive messages that they have seen previously.*

As the chosen slave becomes the master, no messages that are published to the mirrored queue during this time will be lost: messages published to a mirrored queue are always published directly to the master and all slaves. Thus should the master fail, the messages continue to be sent to the slaves and will be added to the queue once the promotion of a slave to the master completes.

Similarly, messages published by clients using **publisher confirms** will still be confirmed correctly even if the master (or any slaves) fail between the message being published and the message being able to be confirmed to the publisher. Thus from the point of view of the publisher, publishing to a mirrored queue is no different from publishing to any other sort of queue.

If you are consuming from a mirrored queue with *noAck=true* (i.e. the client is not sending message acknowledgements) then messages can be lost. This is no different from the norm of course: the broker considers a message *acknowledged* as soon as it has been sent to a *noAck=true* consumer, and should the client disconnect abruptly, the message may never be received. In the case of a mirrored queue, should the master die, messages that are in-flight on their way to *noAck=true* consumers may never be received by those clients, and will not be requeued by the new master. Because of the possibility that the consuming client is connected to a node that survives, the **Consumer Cancellation Notification** is useful in identifying when such events may have occurred. Of course, in practise, if you care about not losing messages then you are advised to consume with *noAck=false*.

Publisher Confirms and Transactions

Mirrored queues support both **Publisher Confirms** and **Transactions**. The semantics chosen are that in the case of both confirms and transactions, the action spans all mirrors of the queue. So in the case of a transaction, a `tx.commit-ok` will only be returned to a client when the transaction has been applied across all mirrors of the queue. Equally, in the case of publisher confirms, a message will only be confirmed to the publisher when it has been accepted by all of the mirrors. It is correct to think of the semantics as being the same as a message being routed to multiple normal queues, and of a transaction with publications within that similarly are routed to multiple queues.

Flow Control

RabbitMQ uses a credit-based algorithm to **limit the rate of message publication**. Publishers are permitted to publish when they receive credit from all mirrors of a queue. Credit in this context means permission to publish. Slaves that fail to issue credit can cause publishers to stall. Publishers will remain stalled until all slaves issue credit or until the remaining nodes consider the slave to be disconnected from the cluster. Erlang detects such disconnections by periodically sending a tick to all nodes. The tick interval can be controlled with the **net_ticktime** configuration setting.

Consumer Cancellation

Clients that are consuming from a mirrored queue may wish to know that the queue from which they have been consuming has failed over. When a mirrored queue fails over, knowledge of which messages have been sent to which consumer is lost, and therefore all unacknowledged messages are redelivered with the `redelivered` flag set. Consumers may wish to know this is going to happen.

If so, they can consume with the argument `x-cancel-on-ha-failover` set to `true`. Their consuming will then be cancelled on failover and a **consumer cancellation notification** sent. It is then the consumer's responsibility to reissue `basic.consume` to start consuming again.

For example (in Java):

```
Channel channel = ...;
Consumer consumer = ...;
Map<String, Object> args = new HashMap<String, Object>();
args.put("x-cancel-on-ha-failover", true);
channel.basicConsume("my-queue", false, args, consumer);
```

This creates a new consumer with the argument set.