CloudAMQP

4 NOVEMBER
ILEC•London

RabbitMQ Summit - 2019

BUY YOUR TICKET

Part 1: RabbitMQ Best Practice

2017-12-29

We have been working with RabbitMQ a long time, and we have probably seen way more configuration mistakes than anybody else. We know how to configure for optimal performance and how to get the most stable cluster. We will in this series share our knowledge!

LOVISA JOHANSSON
Developer

Some applications require really high throughput while other applications are publishing batch jobs that can be delayed for a while. The goal when designing your system should be to maximize combinations of performance and availability that make sense for your specific application. Bad architecture design decisions or client-side bugs, can damage your broker or affect your throughput.

FREE EBOO

Your publisher might be halted or the server might crash due to high memory usage. This article series focuses on best practice for RabbitMQ. Dos and don'ts mixed with best practice for two different usage categories; high availability and high performance (high throughput). We will, among other things, discuss queue size, common mistakes, lazy queues, prefetch values, connections and channels, HiPE and number of nodes in a cluster. Those are all generally best practice rules, based on the experience we have gained while working with RabbitMQ.

"The Optimal RabbitMQ Guide"

Download

Part 2 - Best Practice for High Performance

Instructions for

High Performance

Part 3 - Best Practice for High Availability

Instructions for

High Availability

Part 4 - 13 common RabbitMQ mistakes

RabbitMQ mistakes

Update: Diagnostics Tool for RabbitMQ

We have, due to popular demand, created a diagnostic tool for RabbitMQ that is avaliable for all dedicated instances in CloudAMQP. A link to the tool can be found in the control panel for your instances.

Queues

Keep your queue short if possible

Many messages in a queue can put a heavy load on RAM usage. In order to free up RAM, RabbitMQ start flushing (page out) messages to disk. This process deteriorates queueing speed. The page out process usually takes time and blocks the queue from processing messages when there are many messages to page out. Many messages might affect the performance of the broker negatively.

It is also time-consuming to restart a cluster with many messages since the index has to be rebuilt. It also takes time to sync messages between nodes in the cluster after a restart.

Enable lazy queues to get predictable performance

A feature called *lazy queues* was added in RabbitMQ 3.6. Lazy queues are queues where the messages are automatically stored to disk. Messages are only loaded into memory when they are needed. With lazy queues, the messages go straight to disk and thereby the RAM usage is minimized, but the throughput time will take longer time.

We have seen that lazy queues create a more stable cluster, with better **predictable performance.** Your messages will not, without a warning, get flushed to disk. You will not suddenly be taken by a performance hit. If you are sending a lot of messages at once (e.g. processing batch jobs) or if you think that your consumers will not keep up with the speed of the publishers all the time, we recommend you to enable lazy queues.

Please note that you should disable lazy queues if you require really high performance - if you're queues always are short, or set a max-length policy.

Limit queue size, with TTL or max-length

Another thing that could be recommended for applications that often get hit by spikes of messages, and where throughput is more important than anything else, is to set a max-length on the queue. This keeps the queue short by discarding messages from the head of the queue so that it’s never getting larger than the max-length setting.

Number of queues

Queues are single-threaded in RabbitMQ, and one queue can handle up to about 50k messages/s. You will achieve better throughput on a multi-core system if you have multiple queues and consumers. You will achieve optimal throughput if you have as many queues as cores on the underlying node(s).

The RabbitMQ management interface collects and calculates metrics for every queue in the cluster. This might slow down the server if you have thousands upon thousands of active queues and consumers. The CPU and RAM usage may also be affected negatively if you have too many queues.

Split your queues over different cores

Queue performance is limited to one CPU core. You will, therefore, get better performance if you split your queues into different cores, and also into different nodes if you have a RabbitMQ cluster.

RabbitMQ queues are bound to the node where they were first declared. Even if you create a cluster of RabbitMQ brokers, all messages routed to a specific queue will go to the node where that queue lives. You can manually split your queues evenly between nodes, but the downside is that you need to remember where your queue is located.

We recommend two plugins that will help you if you have multiple nodes or a single node cluster with multiple cores.

Consistent hash exchange plugin
The consistent hash exchange plugin allows you to use an exchange to load-balance messages between queues. Messages sent to the exchange are consistently and equally distributed across many queues, based on the routing key of the message. The plugin creates a hash of the routing key and spread the messages out between queues that have a

binding to that exchange. It could quickly become problematically to do this manually, without adding too much information about numbers of queues and their bindings into the publisher.

The consistent hash exchange plugin can be used if you need to get maximum use of many cores in your cluster. Note that it’s important to consume from all queues. Read more about the consistent hash exchange plugin here.

RabbitMQ sharding
The RabbitMQ sharding plugin does the partitioning of queues automatically for you, i.e., once you define an exchange as sharded, then the supporting queues are automatically created on every cluster node and messages are sharded across them. RabbitMQ sharding shows one queue to the consumer, but it could be many queues running behind it in the background. The RabbitMQ Sharding plugin gives you a centralized place to where you can send your messages, plus load balancing across many nodes, by adding queues to the other nodes in the cluster. Read more about RabbitMQ Sharding here.

Don’t set your own names on temporary queues

Giving a queue a name is important when you want to share the queue between producers and consumers, but it's not important if you are using temporary queues. Instead, you should let the server choose a random queue name instead of making up your own names - or modify the RabbitMQ policies.

Auto-delete queues you are not using

Client connections can fail and potentially leave unused resources (queues) behind, and leaving to many queues behind might affect performance. There are three ways delete a queue automatically.

You can set a **TTL policy** on the queue. E.g., a TTL policy of 28 days deletes queues that haven't been consumed from for 28 days.

An **auto-delete** queue is deleted when its last consumer has canceled or when the channel/connection is closed (or when it has lost the TCP connection with the server).

An **exclusive queue** can only be used (consumed from, purged, deleted, etc) by its declaring connection. Exclusive queues are deleted when their declaring connection is closed or gone (e.g., due to underlying TCP connection loss).

Set limited use of priority queues

Each priority level uses an internal queue on the Erlang VM, which takes up some resources. In most use cases it is sufficent to have no more than 5 priority levels.

Payload - RabbitMQ message size and types

A common question is how to handle the payload (message size) of messages sent to RabbitMQ. You should of course not send very large files in messages, but messages per second is a way larger bottleneck than the message size it self. Sending multiple small messages might be a bad alternative. A better idea could be to bundle them into one larger message and let the consumer split it up. However, if you bundle multiple messages you need to keep in mind that this might affect the processing time. If one of the bundled messages fails - do you need to re-process them all? How you set this up depends on bandwidth and your architecture.

Connections and channels

Each connection uses about 100 KB of RAM (and even more, if TLS is used). Thousands of connections can be a heavy burden on a RabbitMQ server. In the worst case, the server can crash due to out-of-memory. The AMQP protocol has a mechanism called channels that “multiplexes” a single TCP connection. It is recommended that each process only creates one TCP connection, and uses multiple channels in that connection for different threads. Connections should also be long-lived. The handshake process for an AMQP connection is quite involved and requires at least 7 TCP packets (more if TLS is used).

Instead, channels can be opened and closed more frequently, if required. Even channels should be long-lived if possible, e.g. reuse the same channel per thread of publishing. Don’t open a channel for each time you are publishing. Best practise is to reuse connections and multiplex a connection between threads with channels. You should ideally only have one connection per process, and then use a channel per thread in your application.

Don’t share channels between threads

You should also make sure that you don’t share channels between threads as most clients don’t make channels thread-safe (because it would have a serious negative effect on the performance impact).

Don’t open and close connections or channels repeatedly

Make sure that you don’t share channels between threads as most clients don’t make channels thread-safe (because it would have a serious negative effect on the performance impact).

Separate connections for publisher and consumer

Separate connections for publisher and consumer to get high throughput. RabbitMQ can apply back pressure on the TCP connection when the publisher is sending too many messages to the server to handle. If you consume on the same TCP connection the server might not receive the message acknowledgements from the client. Thus, the consume performance will be affected too. And with lower consume speed the server will be overwhelmed.

A large number of connections and channels might affect the RabbitMQ management interface performance

Another effect of having a large number of connections and channels is the performance of the RabbitMQ management interface. For each connection and channel performance, metrics have to be collected, analyzed and displayed.

Acknowledgements and Confirms

Messages in transit might get lost in an event of a connection failure, and such a message might need to be retransmitted. Acknowledgements let the server and clients know when to retransmit messages. The client can either ack the message when it receives it, or when the client has completely processed the message. Acknowledgement has a performance impact, so for fastest possible throughput, manual acks should be disabled.

A consuming application that receives important messages should not acknowledge messages until it has finished whatever it needs to do with them so that unprocessed messages (worker crashes, exceptions etc) don't go missing.

Publish confirm, is the same thing, but for publishing. The server acks when it has received a message from a publisher. Publish confirm also has a performance impact. However, one should keep in mind that it’s required if the publisher needs at-least-once processing of messages.

Unacknowledged messages

All unacknowledged messages have to reside in RAM on the servers. If you have too many unacknowledged messages you will run out of memory. An efficient way to limit unacknowledged messages is to limit how many messages your clients prefetch. Read more about prefetch in the prefetch section.

Persistent messages and durable queues

If you cannot afford to lose any messages, make sure that your queue is declared as “durable” and your messages are sent with delivery mode "persistent".

In order to avoid losing messages in the broker, you need to be prepared for broker restarts, broker hardware failure, or broker crashes. To ensure that messages and broker definitions survive restarts, we need to ensure that they are on disk. Messages, exchanges, and queues that are not durable and persistent will be lost during a broker restart.

Persistent messages are heavier as they have to be written to disk. Keep in mind that lazy queues will have the same effect on performance, even though you are sending transient messages. For high performance - use transient messages.

TLS and AMQPS

You can connect to RabbitMQ over AMQPS, which is the AMQP protocol wrapped in TLS. TLS has a performance impact since all traffic has to be encrypted and decrypted. For maximum performance we recommend using VPC peering instead, then the traffic is private and isolated without involving the AMQP client/server.

At CloudAMQP we configure the RabbitMQ servers to only accept and prioritize fast but secure encryption ciphers.

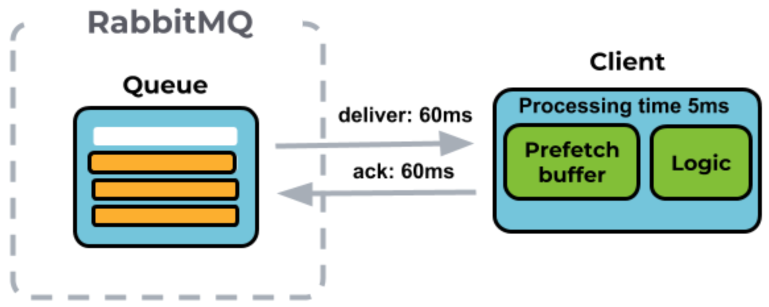
Prefetch

The prefetch value is used to specify how many messages that are being sent to the consumer at the same time. It is used to get as much out of your consumers as possible.

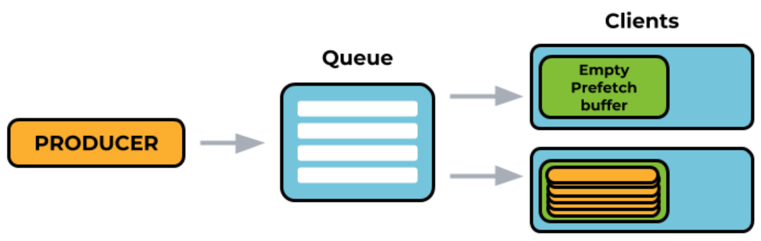
From RabbitMQ.com: *"The goal is to keep the consumers saturated with work, but to minimise the client's buffer size so that more messages stay in Rabbit's queue and are thus available for new consumers or to just be sent out to consumers as they become free."*

RabbitMQ default prefetch setting gives clients an unlimited buffer, meaning that RabbitMQ by default send as many messages as it can to any consumer that looks ready to accept them. Messages that are sent are cached by the RabbitMQ client library (in the consumer) until it has been processed. Prefetch limits how many messages the client can receive before acknowledging a message. All pre-fetched messages are removed from the queue, and invisible to other consumers.

A too small prefetch count may hurt performance since RabbitMQ is most of the time waiting to get permission to send more messages. The image below is illustrating long idling time. In the example, we have a QoS prefetch setting of 1. This means that RabbitMQ won't send out the next message until after the round trip completes (deliver, process, acknowledge). Round time in this picture is in total 125ms with a processing time of only 5ms.



A large prefetch count, on the other hand, could take lots of messages of the queue and deliver it to one single consumer, and keep other consumers in an idling state.



How to set correct prefetch value?

If you have one single or few consumers processing messages quickly, we recommend prefetching many messages at once. Try to keep your client as busy as possible. If you have about the same processing time all the time and network behavior remains the same - you can simply take the total round trip time / processing time on the client for each message, to get an estimated prefetch value.

If you have many consumers, and a short processing time, we recommend a lower prefetch value than for one single or few consumers. A too low value will keep the consumers idling a lot since they need to wait for messages to arrive. A too high value may keep one consumer busy, while other consumers are being kept in an idling state.

If you have many consumers, and/or a long processing time, we recommend you to set prefetch count to 1 so that messages are evenly distributed among all your workers.

Please note that if your client auto-ack messages, the prefetch value will have no effect.

A typical mistake is to have an unlimited prefetch, where one client receives all messages and runs out of memory and crashes, and then all messages are re-delivered again.

For information about RabbitMQ prefetch can be found in this recommended RabbitMQ documentation.

HiPE

HiPE will increase server throughput at the cost of increased startup time. When you enable HiPE, RabbitMQ is compiled at start up. The throughput increases with 20-80% according to our benchmark tests. The drawback of HiPE is that the startup time increases quite a lot too, about 1-3 minutes. HiPE is still marked as experimental in RabbitMQ's documentation.

Don’t enable HiPE if you require high availability.

Number of Nodes in your cluster (Clustering and High Availability)

When you create a CloudAMQP instance with one node you will get one single node with high performance. One node will give you the highest performance since messages don’t need to be mirrored between multiple nodes.

When you create a CloudAMQP instance with two nodes you will get half the performance compared to the same plan size for a single node. The nodes are located in different availability zones and queues are automatically mirrored between availability zones. Two nodes will give you high availability since one node might crash or be marked as impaired, but the other node will still be up and running, ready to receive messages.

When you create a CloudAMQP instance with three nodes you will get 1/4 of the performance compared to the same plan size for a single node. The nodes are located in different availability zones and queues are automatically mirrored between availability zones. You will also get pause minority - by shutting down the minority component you reduce duplicate deliveries as compared to allowing every node to respond. Pause minority is a partition handling strategy in a three node cluster that protects data from being inconsistent due to net-splits.

Remember to enable HA on new vhosts

A common error we have noticed on CloudAMQP clusters is that users create a new vhost but forget to enable an HA-policy for a new vhost. Messages will not be synced between nodes without an HA-policy.

Routing (exchanges setup)

Direct exchanges are the fastest to use. If you have many bindings RabbitMQ has to calculate where to send the message.

Disable unused plugins

Some plugins might be super nice to have, but they might consume a lot CPU or RAM useage. Therefore, they are not recommended for a production server. Make sure to disable plugins that you are not using. You are able to enable many different plugins via the control panel in CloudAMQP.

Do not set RabbitMQ Management statistics rate mode to detailed in production

Setting RabbitMQ Management statistics rate mode to detailed has a serious performance impact and should not be used in production.

Use updated RabbitMQ client libraries

Make sure that you are using the latest recommended version of client libraries. Check our documentation, and feel free to ask us if you have any questions regarding which library to use.

Use latest stable RabbitMQ and Erlang version

Try to stay up-to-date with the latest stable versions of RabbitMQ and Erlang. We usually test new major versions to a great extent before we release them for customers. Please be aware that we always have the most recommended version as the selected option (default) in the dropdown of where you select a version for a new cluster.

Use TTL with caution

Dead lettering and TTL are two popular features in RabbitMQ that should be used with caution. TTL and dead lettering can generate performance effects that you have not foreseen.

Dead lettering

A queue that is declared with the *x-dead-letter-exchange* property will send messages which are either rejected, nacked or expired (with TTL) to the specified dead-letter-exchange. If you specify *x-dead-letter-routing-key* the routing key of the message with be changed when dead lettered.

TTL

By declaring a queue with the x-message-ttl property, messages will be discarded from the queue if they haven't been consumed within the time specified.

GUIDE - RABBITMQ BEST PRACTICE

CONTINUE WITH PART 2
RabbitMQ Best Practice for High Performance (High Throughput)

Start your FREE plan today!

13,000+ users including these smart companies