Sizing your Rabbits

One of the problems we face at the RabbitMQ HQ is that whilst we may know lots about how the broker works, we don't tend to have a large pool of experience of designing applications that use RabbitMQ and which need to work reliably, unattended, for long periods of time. We spend a lot of time answering questions on the mailing list, and we do consultancy work here and there, but in some cases it's as a result of being contacted by users building applications that we're really made to think about long-term behaviour of RabbitMQ. Recently, we've been prompted to think long and hard about the basic performance of queues, and this has lead to some realisations about provisioning Rabbits.

RabbitMQ's queues are fastest when they're empty. When a queue is empty, and it has consumers ready to receive messages, then as soon as a message is received by the queue, it goes straight out to the consumer. In the case of a persistent message in a durable queue, yes, it will also go to disk, but that's done in an asynchronous manner and is buffered heavily. The main point is that very little book-keeping needs to be done, very few data structures are modified, and very little additional memory needs allocating. Consequently, the CPU load of a message going through an empty queue is very small.

If the queue is not empty then a bit more work has to be done: the messages have to actually be queued up. Initially, this too is fast and cheap as the underlying functional data structures are very fast. Nevertheless, by holding on to messages, the overall memory usage of the queue will be higher, and we are doing more work than before per message (each message is being both enqueued and dequeued now, whereas before each message was just going straight out to a consumer), so the CPU cost per message is higher. Consequently, the top speed you'll be able to achieve with an empty queue will be higher than the top speed of a queue with a fixed N messages in it, even if N is very small.

If the queue receives messages at a faster rate than it can pump out to consumers then things get slower. As the queue grows, it will require more memory. Additionally, if a queue receives a spike of publications, then the queue must spend time dealing with those publications, which takes CPU time away from sending existing messages out to consumers: a queue of a million messages will be able to be drained out to ready consumers at a much higher rate if there are no publications arriving at the queue to distract it. Not exactly rocket science, but worth remembering that publications arriving at a queue can reduce the rate at which the queue drives its consumers.

Eventually, as a queue grows, it'll become so big that we have to start writing messages out to disk and forgetting about them from RAM in order to free up RAM. At this point, the CPU cost per message is much higher than had the message been dealt with by an empty queue.

None of this seems particularly profound, but keeping these points in mind when building your application turns out to be very important.

Let's say you design and build your application, using RabbitMQ. There will be some set of publishers, and some set of consumers. You test this, and let's say that in total, in one part of the system, you find that the maximum rate that'll ensure your queues stay empty or very near empty is 2000 msgs/second. You then choose a machine on which to run RabbitMQ, which might be on some sort of virtual server. When testing at 2000 msgs/second you found the CPU load of the box running RabbitMQ was not very high: the bottleneck was elsewhere in the application -- most likely in the consumers of your queues (you were measuring the maximum stable end-to-end performance) -- so RabbitMQ itself wasn't being overly stressed and so wasn't eating up much CPU. Consequently, you choose a virtual server which isn't enormously powerful. You then launch the application and sure enough, everything looks OK.

Over time, your application becomes more popular, and so your rates increase.

Eventually, you get to a point where your consumers are running flat out, and your queues are staying nearly empty. But then, at the most popular time of the day for your application, your publishers push a few more messages into your queues than before. This is just normal growth -- you have more users now and so it's not surprising messages are being published a bit faster than before. What you hope will happen is that RabbitMQ will just happily buffer up the messages and will eventually feed them to your consumers who will be able to work through the back-log during quieter times of the day.

The problem is that this might not be able to happen. Because your queues are now (albeit briefly) receiving more messages than your consumers can cope with, the queues spend more CPU time dealing with each message than they used to when the queue was empty (the messages now have to be queued up). That takes away CPU time from driving the consumers

and unfortunately, as you chose a machine for RabbitMQ which didn't have a great deal of spare CPU capacity, you start maxing out the CPU. Thus your queues can't drive your consumers quite as hard as before, which in turn makes the rate of growth of the queues increase. This in turn starts to push the queues to sizes where they have to start pushing messages to disk in order to free up RAM which again takes up more CPU which you don't have, and by this point, you're likely in deep trouble.

What can you do?

At this immediate point, you need to get your queues drained. Because your queues are spending more time dealing with new messages arriving than with pushing messages out to consumers, it's unlikely that throwing more consumers at the queues is going to significantly help. You really need to get the publishers to stop.

If you have the luxury of just turning off the publishers then do that, and turn them back on when the queues become empty again. If you can't do that, then you need to divert their load somewhere else, but given that your Rabbit is writing out to disk to avoid running out of memory, and is maxing out the CPU, adding new queues to your current Rabbit is not going to help -- you need a new Rabbit on a different machine. If you've set up a cluster, then you could provision new queues on a node of your RabbitMQ cluster that is not so heavily loaded, then attach lots of new consumers to that, and divert the publishers into there. At this point, you'll realise the value of not using the default nameless exchange and addressing queues directly, and instead will be very glad that you had your publishers publish to an exchange you created, thus allowing you to add a new binding to your fresh new queue, and deleting the binding to your old queue, diverting the load, and not having to interrupt your publishers at all. The old queues will then be able to drive their consumers (which you've not removed!) as fast as possible, and the queues will drain. Now in this situation, you have the prospect of messages being processed out of order (new messages arriving in the fresh new queues may be processed by your new consumers before the old messages in the old queues are processed), but if you have multiple consumers off a single queue then you're probably already dealing with that problem anyway.

Prevention, we are often told, is preferable to cure. So how could you design your application to be able to help RabbitMQ cope with these potentially catastrophic situations?

Firstly, don't use a very low `basic.qos` prefetch value in your consumers. If you use a value of 1 then it'll mean that a queue sends out one message to a consumer, and then can't send anything more to that consumer until it receives the acknowledgement back. Only when it's done that can it send out the next message. If it takes a while for the acknowledgement to make its way back to the queue (for example, high latency on the network, or the load that your Rabbit is under may mean it takes a while for that acknowledgement to get all the way through to the queue) then in the meantime, that consumer is sitting there idle. If you use a `basic.qos` prefetch value of 20, for example, then the broker will ensure that 20 messages are sent to the consumer, and then even as the acknowledgement for the first message is (maybe slowly) making its way back to the queue, the consumer still has work to be getting on with (i.e. the next 19 messages). Essentially, the higher the prefetch value, the greater insulation the consumer has from spikes in the round trip time back to the queue.

Secondly, consider not acknowledging every message, but instead acknowledging every N messages and setting the `multiple` flag on the acknowledgement. When the queue is overloaded, it's because it has too much work to do (profound, I know). As a consumer, you can reduce the amount of work it has to do by ensuring you don't flood it with acknowledgements. Thus, for example, you set the `basic.qos` prefetch to 20, but you only send an acknowledgement after you've processed every 10 messages and you set the `multiple` flag to true on the acknowledgement. The queue will now receive one-tenth of the acknowledgements it would have previously received. It will still internally acknowledge all ten messages, but it can do it in a more efficient way if it receives one acknowledgement that accounts for several messages, rather than lots of individual acknowledgements. However, if you're only acknowledging every N messages, be sure that your `basic.qos` prefetch value is higher than N. Probably at least 2*N. If your prefetch value is the same as N, then your consumer will once again be left idle whilst the acknowledgement makes its way back to the queue and the queue sends out a fresh batch of messages.

Thirdly, have a strategy to pivot the load to other queues on others machines if the worst comes to the worst. Yes, it is a good idea to use RabbitMQ as a buffer which insulates publishers from consumers and can absorb spikes. But equally, you need to remember that RabbitMQ's queues go fastest when they're empty, and we always say that you should design your application so that the queues are normally empty. Or to put it another way, the performance of a queue is lowest when you likely need more than ever for it to absorb a large spike. The upshot of that is that unless you test to makes sure you know it will recover, you might be in for a surprise should a non-trivial spike occur which substantially increases the length of your queues. You may not have factored into your thinking that in such a situation, your existing consumers may end up being driven more slowly than before, simply because your queues are busy doing other things (enqueuing messages), and that this can cause a vicious cycle that eventually results in a catastrophic loss of performance of the queue.

The nub of the issue is that a queue is a single-threaded resource. If you've designed your routing topology in such a way that it already does, or at least can, spread messages across

multiple queues rather than just hammering all messages into a single queue, then you're far more likely to be able to respond to such problems occurring quickly and easily, by diverting load and being able to take advantage of additional CPU resources which ensure that you can minimise the CPU-hit per message.

This entry was posted on Saturday, September 24th, 2011 at 12:37 pm by Matthew Sackman and is filed under **HowTo**, **Introductory**. You can follow any responses to this entry through the **RSS 2.0** feed. Both comments and pings are currently closed.

Comments are closed.
