« **High Availability in RabbitMQ: solving part of the puzzle**
Ponies, Dragons and Socks »

Performance of Queues: when less is more

Since the *new persister* arrived in RabbitMQ 2.0.0 (yes, it's not so *new* anymore), Rabbit has had a relatively good story to tell about coping with queues that grow and grow and grow and reach sizes that preclude them from being able to be held in RAM. Rabbit starts writing out messages to disk fairly early on, and continues to do so at a gentle rate so that by the time RAM gets really tight, we've done most of the hard work already and thus avoid sudden bursts of writes. Provided your message rates aren't too high or too bursty, this should all happen without any real impact on any connected clients.

Some recent discussion with a client made us return to what we'd thought was a fairly solved problem and has prompted us to make some changes. First, we **took some time to understand better how CPU use per message varies as queues get longer and go to disk**, and what the effect of that is. The conclusions are not necessarily obvious. Then we started thinking about the justifications behind some of the decisions queues take in the process of going from a purely-in-RAM queue to a purely-on-disk queue.

An AMQP queue in Rabbit is not simple functional FIFO queue. There are in fact at least four "queues" used internally by each AMQP queue, each of which are allowed to hold messages in various different states. These states are things like: Is the message held in RAM (regardless of whether it has *additionally* been written to disk)?; Is the message itself on disk, but its location within the queue still only held in RAM? That sort of thing. Each message in the AMQP queue will only appear at any one time in one of these internal queues, but they can move from queue to queue if and when their state changes (though the movement between these internal queues respects the overall order of messages within the AMQP queue). There is then a fifth "queue" which is not really a queue at all. It is more a couple of numbers that indicate the range of messages that are held solely on disk (if you like, these are pointers to the head and tail of the "queue" which is solely on disk). Messages in this form have, in theory, zero RAM-cost (depending on how you count (numbers cost RAM too you know!), and elsewhere in Rabbit you can be fairly sure the best you can get down to is a few bytes per message). The full gory details can be gleaned from the essay at the top of the **variable_queue** module. It's not really that terrifying, but it isn't exactly noddy stuff either. The tricky bits are working out how you decide which messages should be in which states, and when, and I'm not going to cover those decisions in this post.

A message that is captured by this fifth "queue" will potentially take two reads to go from that purely-on-disk form right back to a fully-in-RAM message. This is because every message has a message ID (which is random, unordered, and allocated to each message before it arrives at any queue, so is useless for determining relative position within an AMQP queue), and within each AMQP queue each message is known by its per-queue sequence ID, which enforces the relative order of messages within an AMQP queue. This fifth "queue" can be thought of as a mapping from sequence ID to message ID (plus some per-message-per-queue state), and then you can use a different subsystem to transform that message ID to the actual message.

As a result of these two reads (though the way we structure it, one of the reads is shared between 16k messages, so it's probably closer to 1+(1/16384) reads per message, at least by default), we've previously tried to prevent the use of this fifth "queue": in the past, even when memory is running very low, we'll write messages out to disk fully, but then still hold onto a record in RAM (though by this point it's a fairly small record per message), assuming that this will give us an advantage later on: yes, it costs more RAM, but if some other big AMQP queue suddenly gets deleted and frees up lots of RAM then by holding onto this smallish record per message, we avoid having to do the two reads to go from the fifth "queue" back to a full message, and will only have to do a single read instead. Only when RAM runs out completely will we suddenly dump (almost) everything into this fifth "queue" (though by this point, everything will be on disk by now anyway so it's more or less a no-op -- we're just freeing up RAM in this transition).

However, because of the effective amortisation of at least one of these reads, using the fifth "queue" isn't as expensive as we feared. Plus, if you start to use it earlier then the queue grows in RAM at a slower rate: messages have the lowest per-message RAM cost when they're in this fifth "queue", and thus the greater your use of this "queue", the lower the rate at which your queue will consume RAM. This on its own helps Rabbit smooth out the transition to purely on disk operation (given the same growth rate in messages, a lower growth rate in RAM will result in a lower rate of disk ops).

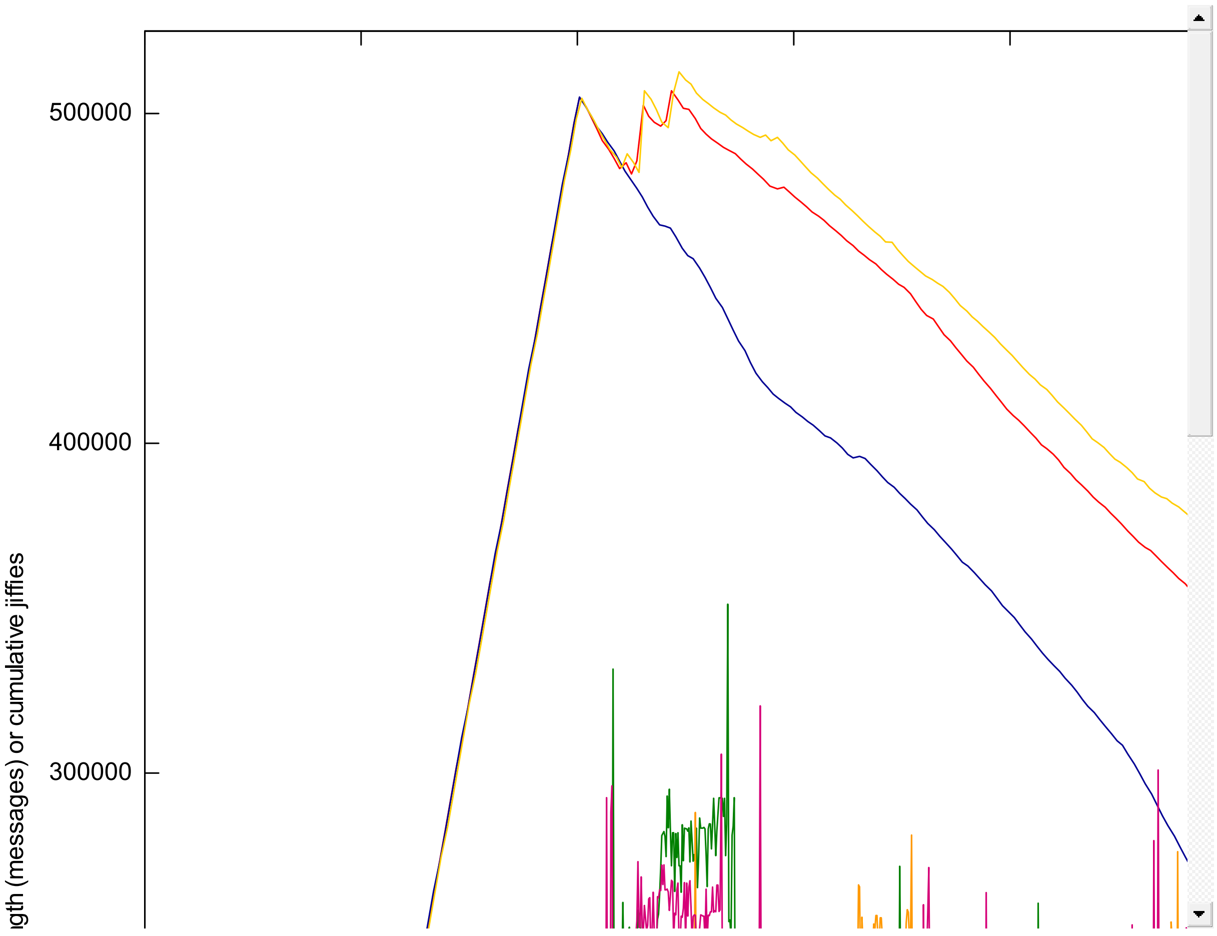
So we've changed the behaviour of Rabbit's AMQP queues to use this fifth "queue" more eagerly. Benchmarks suggest that this seems to result in an AMQP queue's memory use flat-lining earlier on in its growth, and actually seems to make Rabbit able to deliver messages from large AMQP queues out to consumers faster (probably because by limiting the size of the other four internal queues, some operations which were found to be very inefficient (such as joining two functional queues together (Erlang's default queue module does a plain append (++) here, which is expensive)) are avoided, thus more CPU is available for driving consumers). The downside is that queues now spend more time doing reads, but that seems to have been more than offset by the lower user jiffies per message.

Below is a graph. This is very exciting -- not just because of the fact that most of my blog posts are endless words. It shows three runs of the same test program. This test program does the following:

- It creates 3 queues.
- It binds those 3 queues to a fanout exchange.
- It then starts publishing 200-byte messages to that exchange at 600 messages / second.
- For the first 120 seconds, it has 20 consumers per queue consuming without auto-ack, one ack per message, and a QoS pre-fetch of 1. This is known to be a very expensive way of consuming messages. Further, the ack is deliberately delayed so that, ignoring network latency, the maximum aggregate consuming rate will be 1200 messages / second.
- After 120 seconds, the consumers are stopped, and not started again until there is a total backlog of 500,000 messages (i.e. each of the 3 queues will have around 166,000 messages).
- After that, the consumers resume as before, and you hope that the queues can cope with the continuing publishes and drive their backlog out to the consumers. Hopefully, all the queues will eventually become empty again.

Now depending on your CPU, RAM, network latency, and *high_watermark* settings, this backlog can be purely in RAM only, and so no disk operations ever take place; or it could be purely on disk; or anywhere in between. Our desktops in the office here tend to be too powerful for this test to cause any problems (the backlog always drains), but on some EC2 hosts, with older versions of Erlang and older versions of Rabbit, it was possible to reach a point where this backlog would never drain, and instead grow.

In the following graph, we have three successful runs of this test, both on *m1.large* EC2 instances, with the test being run on a separate EC2 instance (i.e. we really were going across the network). These are running Ubuntu images, but with a locally compiled Erlang R14B04 installation. The three runs are: 1) what was on our default branch prior to this work being merged; 2) our default branch after this work was merged; 3) the 2.6.1 release.



Since 2.6.1 was released, a fairly large number of performance tweaks have been made, and this is shown by the faster rate at which the backlog disappears. But the "default prior to change"

and "2.6.1" memory usages are fairly similar, whereas, on average, the "default post change" has a lower memory usage. The memory measurements are not especially compelling however as, owing to Erlang being an auto-garbage-collected language, it is not always the case that improving memory efficiency internally results in the VM requesting less RAM or returning RAM to the OS faster. What is more compelling is the faster rate at which the backlog is eliminated and the lower cumulative jiffies: even though "default post change" is doing more messages per second than either of the other runs, it still uses fewer jiffies per second than the other runs.

Hopefully this change will make improvements to many users across many scenarios. It's possible there may be some use cases where it performs worse -- we certainly can't rule that out. No problem in software engineering that's worth solving has a single correct solution. This case shows that sometimes, using less memory and doing apparently more disk ops can actually make things go faster overall.

Tags: **performance**, **queues**

This entry was posted on Thursday, October 27th, 2011 at 5:45 pm by Matthew Sackman and is filed under **New Features**. You can follow any responses to this entry through the **RSS 2.0** feed. Both comments and pings are currently closed.

Comments are closed.