Agocs

Christopher Agocs

Tips for Using RabbitMQ in Go

Corrections:

- 4% != .004% : When I was writing the article, my brain translated 99996 into 96000. Big difference. It turns out that I'm unable to dequeue somewhere between .004% and .20% of messages in about half of test runs.

Note:

I've been chatting with some very helpful RabbitMQ-knowledgeable people, and they have some suggestions for the issues I'm seeing that I'm going to check out. I will update this article with my findings.

I want to thank Alvaro Videla and Michael Klishin for reading my first attempt at this post and suggesting different avenues to explore.

Introduction

For the two of you who don't know, RabbitMQ is a really neat AMQP-compliant queue broker. It exists to facilitate the passing of messages between or within systems. I've used it for a couple of different projects, and I've found it to be tremendously capable: I've seen a RabbitMQ instance running on a single, moderately sized, VM handle almost 3GB/s.

I was doing some load testing with RabbitMQ recently, and I found that, if I started attempting to publish more than around 2500 10KB messages per second, ~~about 4%~~ as much as 0.2% of those messages wouldn't make it to the queue during some test runs. I am not sure if this is my code's fault or if I am running into the limits of the RabbitMQ instance I was testing against (probably the former), but with the help of the RabbitMQ community, I was able to come up with some best practices that I've described below.

The examples below are all in Go, but I've tried my best to explain them in such a way that people who are not familiar with Go can understand them.

Terminology

If you're unfamiliar with AMQP, here's some terminology to help understand what's possible with a queue broker and what the words mean.

- **Connection**: A connection is a long-lived TCP connection between an AMQP client and a queue broker. Maintaining a connection reduces TCP overhead. A client can re-use a connection, and can share a connection among threads.
- **Channel**: A channel is a short-lived sub-connection between a client and a broker. The client can create and dispose of channels without incurring a lot of overhead.
- **Exchange**: A client writes messages to an exchange. The exchange forwards each message on to zero or more queues based on the message's routing key.
- **Queue**: A queue is a first-in, first out holder of messages. A client reads messages from a queue. The client can specify a queue name (useful, for example, for a work queue where multiple clients are consuming from the same queue), or allow the queue broker to assign it a queue name (useful if you want to distribute copies of a message to multiple clients).
- **Routing Key**: A string (optionally) attached to each message. Depending on the exchange type, the exchange may or may not use the Routing Key to determine the queues to which it should publish the message.
- **Exchange types**:
  - **Direct**: Delivers all messages with the same routing key to the same queue(s).
  - **Fanout**: Ignores the routing key, delivers a copy of the message to each queue bound to it.
  - **Topic**: Each queue subscribes to a topic, which is a regular expression. The exchange delivers the message to a queue if the queue's subscribed topic matches the message.
  - **Header**: Ignores the routing key and delivers the message based on the AMQP header. Useful for certain kinds of messages.

Testing methodology

Here is the load tester I wrote: https://github.com/backstop/rabbit-mq-stress-tester. It uses the streadway/amqp library. Per this issue, my stress tester does not share connections or channels between Goroutines — it launches a configurably-sized pool of Goroutines, each of which maintains its own

connection to the RabbitMQ server.

To run the same tests I was running:

- Clone the repo or install using `go get github.com/backstop/rabbit-mq-stress-tester`
- Open two terminal windows. In one, run

```
./tester -s test-rmq-server -c 100000
```

That will launch the in Consumer mode. It defaults to 50 Goroutines, and will consume 100,000 messages before quitting.

- In the other terminal window, run

```
./tester -s test-rmq-server -p 100000 -b 10000 -n 100 -q
```

This will run the tester in Producer mode. It will (-p)roduce 100,000 messages of 10,000 (-b)ytes each. It will launch a pool of 100 Goroutines (-n), and it will work in (-q)uiet mode, only printing NACKs and final statistics to stdout.

What I found is that, roughly half the time I run the above steps, the consumer will only consume 99,000 and change messages (typically greater than 99,980, but occasionally as low as 99,800). I was unable to find any descriptive error messages in the `rabbitmq@test-rmq-server.log` file.

I can change that, though. If I run the producer like this:

```
./tester -s test-rmq-server -p 100000 -b 10000 -n 100 -q -a
```

then each Goroutine waits for an ACK or NACK from the RabbitMQ server before publishing the next message (that's what the -a flag does). I have never seen a missing message in this mode. The functionality of the -a flag is described in the next section.

Some things that I don't think are the culprit:

- Memory-based flow control: Memory usage as reported by `top` never exceeds approximately 22%. Also, no messages in the log file.
- Per-connection flow control: After fussing with `rabbitmqctl list_connections` for a while, I was not able to find evidence of a connection that had been blocked. I'm not sure of these results, though, so if someone would be willing to give me a hand with this, that would be awesome.

Ensuring your message got published

Like I said earlier, I was doing some stress testing against a RabbitMQ instance and a small number of messages that I attempted to publish did not get dequeued. I reached out to the RabbitMQ community, and someone on their IRC channel told me to look up Confirm Select.

When you place a channel into Confirm Select, the AMQP broker will respond with an ACK with a for each message passed to it on that channel. Included with the ACK is an integer that increments with each ACK, similar to a TCP sequence ID. If something goes wrong, the broker will respond with a NACK. In Go, placing a channel into Confirm Select looks like this:

Putting a channel in Confirm Select

```
channel, err := connection.Channel()
if err != nil {
  println(err.Error())
  panic(err.Error())
}
channel.Confirm(false)

ack, nack := channel.NotifyConfirm(make(chan uint64, 1), make(chan uint64, 1))
```

The above `channel.Confirm(false)` puts the channel into Confirm mode, and the `false` puts the client out of NoWait mode such that the client waits for an ACK or NACK after each message. `ack` and `nack` are golang `chans` that receive the integers included with the ACKs or NACKs. If you were in NoWait mode, you could use them to bulk publish a bunch of messages and then figure out which messages did not make it.

Listening for the ACK looks like this:

Publish a message and wait for confirmation

```
    channel.Publish("", q.Name, true, false, amqp.Publishing{
       Headers:         amqp.Table{},
       ContentType:     "text/plain",
       ContentEncoding: "UTF-8",
       Body:            messageJson,
       DeliveryMode:    amqp.Transient,
       Priority:        0,
   },
   )

   select {
   case tag := <-ack:
       log.Println("Acked ", tag)
   case tag := <-nack:
       log.Println("Nack alert! ", tag)
   }
```

After each publish, I'm performing a read off of the `ack` and `nack` chans (that is what `select` does). That read blocks until the client gets an ACK or NACK back from the broker.

The above examples are in Go, but there's an equivalent in the other libraries I've played with. Clojure (langohr) has `confirm/select` and `confirm/wait-for-confirms-or-die`.

Can we do better?

Yes. Rather than wait for an ACK after each publish, it's better to publish a bunch of messages, listen for ACKs, and then handle failures. I didn't, because I was already seeing performance several orders of magnitude better than I needed.

We can also wrap blocks of messages in a transaction if we need to ensure that all messages get published and retain order, but doing that incurs something like a 250x performance penalty.

Pool your Goroutines and avoid race conditions

This issue proved interesting (if ultimately not relevant to my problem). It looks like the person who filed the issue was running into two issues:

- There was a race condition in the code that counted ACKs / NACKs
- The one-Goroutine-per-publish strategy causes a condition where the 2000 goroutines waiting for network IO prevent the goroutine listening for ACKs / NACKs from receiving sufficient CPU cycles.

I got around this in two ways: I have a fixed-size pool of Goroutines performing the publishing, and each goroutine handles its own Publish –> Ack lifecycle.

Ensuring messages get handled correctly

A message queue is of little use if messages just sit there, so it is prudent to include a consumer or two. But, what happens if your consumer crashes? Does your message get lost in the ether?

The answer is **AutoAck**. More specifically, realizing that AutoAck is dangerous and wrong.

When a consumer consumes a message from a queue, the queue broker waits for an ACK before discarding the message. When a consumer has AutoAck enabled, it sends the ACK (thus causing the message to be discarded) instantly upon receiving the message. It's smarter to read the next message on the queue, handle the message properly, and then send the ACK.

Reading and acknowledging messages

```
1  autoAck := false
2
3  msgs, err := channel.Consume(q.Name, "", autoAck, false, false, false, nil)
4  if err != nil {
5    panic(err)
6  }
```

```
7
8  for d := range msgs { // the d stands for Delivery
9    log.Printf(string(d.Body[:])) // or whatever you want to do with the message
10   d.Ack(false)
11 }
```

In the example above, `autoAck` is set to false. Every time I read a message (in the `for d := range msgs` loop), I send an ACK for that message. If I were to call `d.Ack(true)`, that would send an ACK for that message and all previous unacknowledged messages.

If my consumer quits without acknowledging a message, that message is repeated to the next consumer to come by.

Performant results

So, what kind of performance am I getting?

The following numbers are all time to publish and consume 100,000 messages, each with a 10KB payload. The tester was running on my Macbook, and RabbitMQ was running on a Cloudstack VM.

- With Confirm Select
  - Publishing: 24.42s
  - Consuming: 26.79s
- Without Confirm Select
  - Publishing: 16.32s
  - Consuming: 26.13s

The point is, RabbitMQ is fast and Go is fast. When we use one to stress test the other, messages get lost somewhere. If we take a little bit of time to ensure that messages get published and processed properly, we can prevent pesky data loss issues.

Aug 19th, 2014
RabbitMQ, golang, hack, tips

Comments

Join the discussion…

cirpo • 2 years ago

Interesting post, do you have any updates?

1 • Reply • Share ›

Christopher Agocs Mod ➚ cirpo • 2 years ago

Nothing too new, but if you want to see some more code examples and hear questions, check out this talk I gave: http://agocs.org/blog/2014/12/...

• Reply • Share ›

Dave • a year ago

I have a question about RabbitMQ Performance. So is it better to have 100 Queues for 100 Clients when some clients get the same data and the others clients get not the same.Or is it better to send all Messages over one Queue and filter the Data on the Golang RMQ Client? At the moment I send all over one queue and filter the data after reading the queue. But in this case it seems that acknowledgements are unneccssary because it exsits one "RMQ-Receiver" for 100 clients... But Iam not sure how many Queues RMQ can handle at same time and sending perhaps 100 Messages per second to many Queues... I am afraid of getting performance loss when RMQ handles a lot of Queues.

• Reply • Share ›

Christopher Agocs Mod ➚ Dave • a year ago

I would imagine that your network traffic is a bigger bottleneck than the load on your queue broker. If I were you, I'd publish messages to a RabbitMQ exchange with routing keys, and have my clients subscribe to topics on that exchange. That way, your queue broker isn't sending 100 messages for every message it gets.

I honestly don't know the upper limit to the number of topics one RMQ instance can handle. My guess would be that it varies based on how powerful the machine that it's running on is. My gut instinct is that, even for a modest machine, the answer is "well over a hundred".

It would be interesting to see the performance difference between these two methods. If you'd like, it might be neat to work together on this.

• Reply • Share ›

Ezequiel Moreno • a year ago

Hi,

Any hint on how many goroutines per consumer?

Thanks

• Reply • Share ›

Christopher Agocs Mod ➚ Ezequiel Moreno • a year ago

I think we were using one routine per consumer. That didn't appear to be a bottleneck.

• Reply • Share ›

Ezequiel Moreno ➚ Christopher Agocs • a year ago

Thanks Christopher =) , gonna do some stress testing over here. =)

• Reply • Share ›

zdouglas • 2 years ago

As a shot in the dark with your 0.2% failure rate:

> OpenSSH 6.5 and 6.6 have a bug that causes ~0.2% of connections using the
> "curve25519-sha256@libssh.org" KEX exchange method to fail when connecting
> with something that implements the specification correctly.
> OpenSSH 6.7 disables this KEX method when speaking to one of the affected versions.
via: http://www.openbsd.org/56.html

Maybe, if you're using it, upgrading to OpenSSH 6.7 could help?

• Reply • Share ›

Christopher Agocs Mod ➚ zdouglas • 2 years ago

That's a good thought. I ran the test from my macbook onto a virtual machine, though, and here are my ssh versions:

christopheragocs$ ssh -v
OpenSSH_5.9p1, OpenSSL 0.9.8y 5 Feb 2013

root@test-rmq:~# ssh -v
OpenSSH_5.9p1 Debian-5ubuntu1.4, OpenSSL 1.0.1 14 Mar 2012

So I don't think that's it either. Thanks, though. I appreciate the comment.

• Reply • Share ›

zdouglas ➚ Christopher Agocs • 2 years ago

No problem at all! Thank you for the post!

• Reply • Share ›

ALSO ON **AGOCS.ORG**

Sun Visor Bracket | Agocs

8 comments • 2 years ago•

Christopher Agocs — I wish. I did all this at one of the local hackerspaces, Pumping Station One.That said, when Sabrina and I start looking to buy a home, I'll probably be asking a lot of

Projects | Agocs

3 comments • 2 years ago•

Christopher Agocs — Alright smart guy. =)

Ensuring your RabbitMQ messages make it | Agocs

2 comments • 2 years ago•

Christopher Agocs — You're absolutely right. I'm going to investigate these and update.

End Grain Cutting Board | Agocs

1 comment • a month ago•

Warren — yay for hackerspace toolsboo they were not in amazing repair / current maintenance :|great looking cutting board either way :)