Scaling Graphite Clusters

Johan van den Dorpe

My talk tonight is about scaling graphite clusters

Contents

- About graphite
- Scaling python graphite
- carbon-c-relay
- go-carbon

I’m going cover a bit of background about what graphite it and how you can scale the stock python graphite

Over the past few years there has emerged a number of projects that replace graphite components, written in languages other than python. Late last year we deployed carbon-c-relay and go-carbon as we scaled up our cluster to handle an increased number of metrics, so I’m going to discuss how those new components fit into a graphite deployment and what we needed to tune to get them running effectively.

Graphite

- Carbon - daemon that listens to TCP port for stream of time-series data and writes to disk
- Whisper - Flat file database format for time series data
- Graphite Web - graph composer & API returning metric data by querying whisper data and carbon caches
- Originally written by Orbitz

Carbon - daemon that listens to TCP port for stream of time-series data and writes to disk

• Whisper - Flat file database format for time series data
• Graphite Web - graph composer & API returning metric data by querying whisper data and carbon in memory caches
• Originally written by Orbitz about 10 years ago

Sending metrics to graphite

- Easy to get data in:
  - `echo "sample.metric.name 3000 $(date +%s)" | nc graphite 2003`
- Insert data out of order
- Configurable resolution

There are a number of things about graphite that made it a huge improvement over previous systems.

Firstly, it's simple to get data into the system - here's an example. Open a socket to the server, and send a metric name, value and then a timestamp followed by a newline.

Also it's possible to insert data out of order - so you can pull historical data from another source and put it into graphite.

Then, for each metric, you can define the resolution of the data - that is how often we expect a metric to be updated and how frequently we draw points in a graph. This can be as little as 1 second, and as large are you like.

Whisper

- Database format
- A fixed size .wsp file is created per unique metric
- Dot separated metric names are written out as a directory hierarchy, i.e. `sample.metric.name` becomes `sample/metric/name.wsp`
- Each whisper file can have multiple retentions/resolutions i.e. `10s:7d,60s:90d,15m:1y`

Whisper is a file based database format. Each metric ends up as a file on disk, pre allocated with the size required for the configured resolution and retention policies.

Metrics become a hierarchical directory structure. This allows you to create conventions around metric naming, for example services.elasticsearch.cluster name.host name etc. and then use unix shell style globs to select time series when querying.

Retention resolution can be tiered. For instance, you could store high resolution metrics @ every 10 seconds for 7 days and lower resolution metrics @ every 60 seconds for 90 days. So you can view very precise information for a recent event, but still be able to graph long term trends and keep metric data for a long time. This is a major advantage of graphite that other monitoring solutions don’t yet do as well.

Graphite Web

- Rapid prototyping of graphs
- Anyone can create graphs
- Explore data by combining, aggregating and performing calculations on timeseries

The cache and whisper are about receiving and storing metrics, Graphite web is how data is queried. It's a django app, that can output PNG or SVG charts, or return metric data in multiple formats - JSON, CSV or as data you can pass directly to a js library such as rickshaw or dygraph.

It reads metric data from multiple sources - whisper files on disk, metrics in the in-memory cache of carbon, or from another instance of graphite web. This last feature allows a graphite web instance to be a single frontend for a federated cluster of graphite servers.

Once you've stored your data, graphite's graph composer is extremely good at prototyping graphs. Anyone can create a new graph and you can overlay multiple metrics, compare them, combine them, and perform all sorts of calculations on them.

The graph composer is rarely used these days - graphite web just becomes an API for requesting metric data, with grafana generally used as the dashboard tool to display the graphs

Graphite components summary

- Feed data into carbon
- Persist data on-disk with whisper database files
- Query data with graphite web

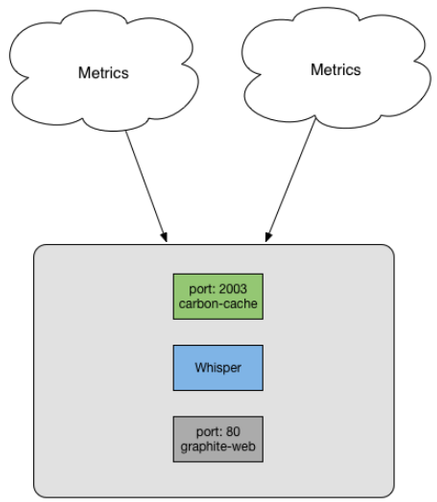
So in summary, read the bulletpoints

Breaking out components

- Multiple carbon-cache daemons
- Multiple cache nodes
- n+1 all components

The process of scaling up a graphite cluster is about breaking out the individual components, and replicating or sharding the data between them. So let’s go over the steps you may take as a cluster grows.

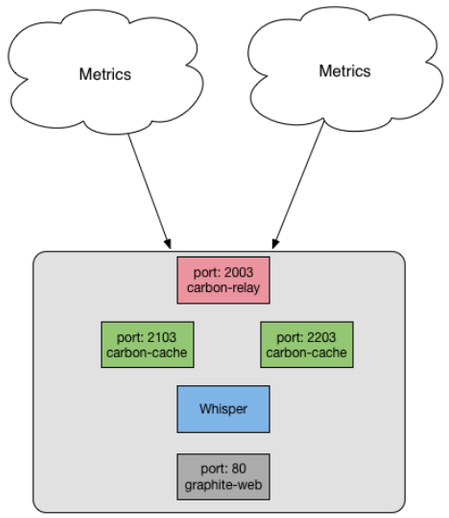
Single Server



At first, the simplest solution is a single server running everything. You run the carbon cache daemon, it receives incoming metrics, those metrics are written to a local filesystem, and you query metrics using the webapp on the server.

Ignoring the lack of any redundancy, and assuming you've got enough disk IO and RAM to support the rate of metrics being received, the likely first problem as the number of metrics you collect increase is carbon-cache using 100% CPU and dropping metrics. This is because it's a single threaded process.

Multiple carbon-cache daemons

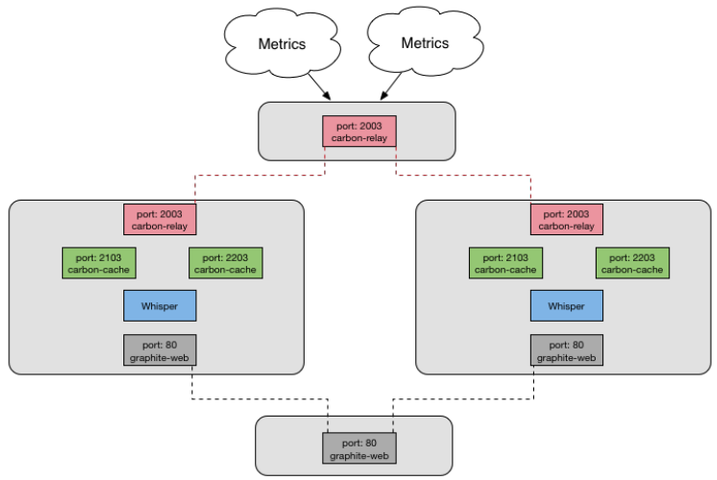


So, the next step is to run multiple carbon cache processes, so you can spread the workload out across multiple CPU cores.

In this scenario you run carbon-relay in front of your carbon cache daemons. carbon-relay is a router for incoming metrics - it will listen for incoming metrics on a defined port, and then forward them on via TCP - so the receiver could potentially be on another host. It can be configured to replicate the metrics, so sending the same metric to multiple receivers, or to shard the received metrics up across multiple receivers, which is what we will do here. Usually this is done using consistent-hashing mode, which allows a hands off and mostly even distribution between receivers.

The relay also supports some level of caching in case the downstream process can’t keep up with the rate of metrics being ingested.

Multiple cache servers

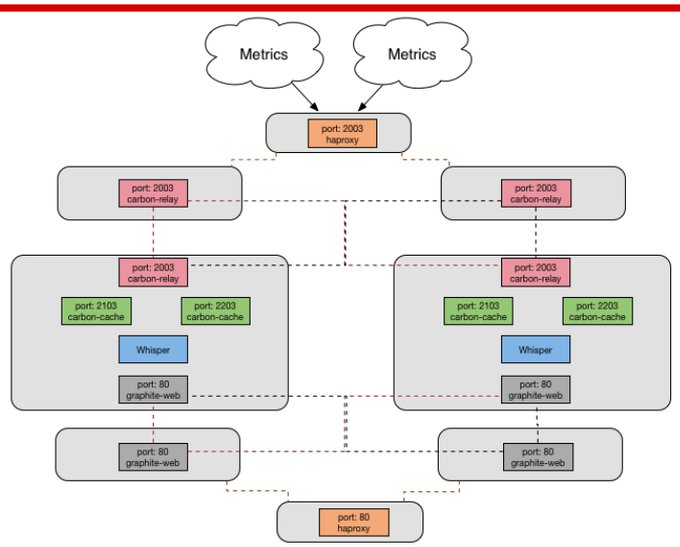


So the next step is either you have run out of capacity on one server, so you want to distribute your metrics across multiple nodes. Or, you want to have a HA setup and keep the same data on two servers. This is the sort of solution I'm striving for - scaling the individual nodes as far as possible, but not relying on any one of them for the whole system to operate.

So now you add more nodes - on ingest you have a dedicated node running carbon-relay, which fans out the metrics across your two cache nodes. Each cache node is then configured the same as in the previous example.

You also will need to add a dedicated web node. This provides a federated view of your multiple servers, communicating with the cache backends and presenting a single result for queries. If you run multiple cache nodes with the same data, this web frontend will merge the results - so if for example a node was offline for some time and the timeseries from that server have gaps, it will fill the gaps from the results from the other cache node.

Multiple Relays, Web Hosts



If you’re looking for HA for your cluster, then having a single relay and web host isn’t satisfactory. So we would add a TCP load balancer like haproxy in front of multiple identically configured carbon relay servers, and a HTTP load balancer in front of multiple web servers.

Of course haproxy is now the single point of failure, but there are various approaches to handle that depending on your environment. We are using a cluster of haproxy nodes, and BGP to route traffic to the active cluster members.

So this looks pretty much like our cluster a year ago, except we had more than two of some components - we were running at least 4 carbon cache daemons but occasionally some of those instances were hitting 100% CPU with our workload. We could keep scaling that out but that couldn’t go on forever on a single box solution.

We also had haproxy running on each of the relay and cache nodes as the first point to ingest metrics - sometimes the relays would crash, so we wanted to run multiples of them too. So we had even more components than shown here.

carbon-c-relay

- Lower resource consumption
- Useful statistics
- Regex based conditional forwarding to receivers
- Supports graphite consistent hashing
- Can support larger cache, receivers can be down longer before losing metrics

So we started looking at the other projects to replace the python graphite components. First we looked at carbon relay replacements and we settled on carbon-c-relay. It supports all the same features as carbon-relay, but with lower resource consumption since it's written in C, and it has better regex based conditional forwarding to receivers.

The biggest benefit of changing to carbon-c-relay is it's larger cache.

Tuning carbon-c-relay

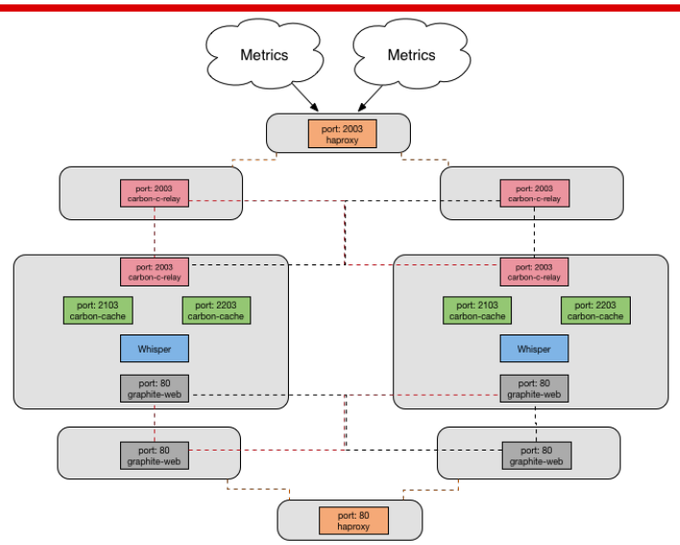
- queuesize - default 2500 too low, 5million+
- backlog - default 32, we use 128

These were the only tuning parameters we needed to change:

queuesize - Each configured server that the relay sends metrics to has a queue associated with it. This queue allows for disruptions and bursts to be handled. Constraining the queue size limits memory consumption, but setting it too low risks you losing metrics. We have set it to 5 million, this has provided enough capacity that if you need to perform a short maintenance on a cache node involving a reboot, then we don’t drop any metrics. Previously with carbon-relay this wasn’t the case.

backlog sets the TCP connection listen backlog. If you have a lot of concurrent connections you need to increase this to avoid connection refused errors on clients

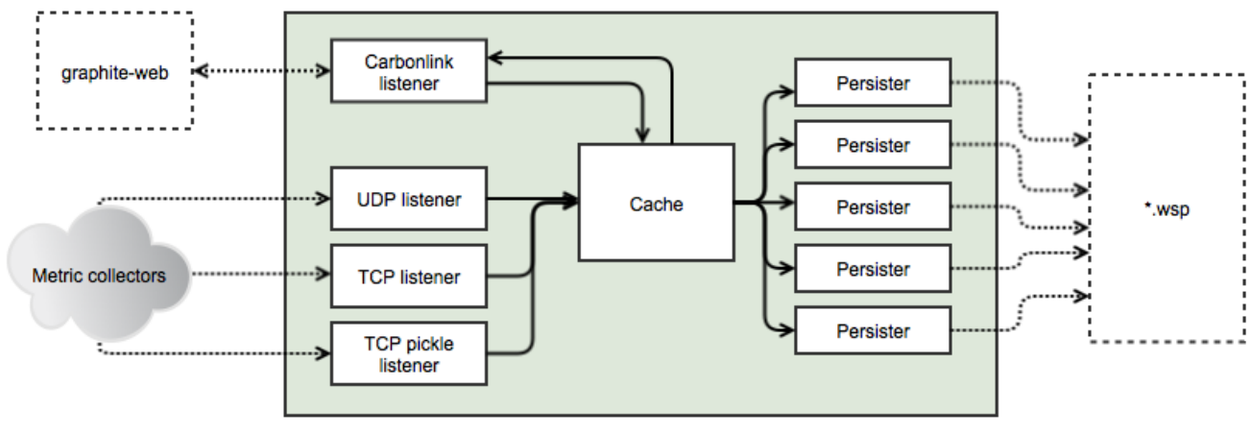
carbon-c-relay



So this is the updated cluster architecture after replacing carbon relay with carbon-c-relay - you can see it doesn’t change our architecture much.

It does remove haproxy on the relay node, but the benefits come from the stability, lower resource consumption, and more powerful rules system for forwarding metrics, and the larger cache.

go-carbon



On the cache nodes we evaluated and decided to implement go-carbon. go-carbon is a Golang implementation of Graphite/Carbon server with the same internal architecture: Agent -> Cache -> Persister

The challenge with setting up go-carbon is piecing together an appropriate architecture and configuration for your needs - most of the information for this is hidden in github issues for the project.

go-carbon

- Lower resource consumption
- Single process, multiple worker threads
- Receive metrics from TCP/UDP, HTTP and Kafka
- Carbonlink support (query in memory cache)
- On restart, dump in memory cache and reload

go carbon proved to be a good choice because it has lower resource consumption, and also it’s a single multithreaded process - this removes the config sprawl required when running so many single-threaded relay and cache processes.

It supports all the same input methods as python carbon, plus a few more, and supports that carbonlink protocol that allows graphite-web to query the cache. And it has some neat features, like persisting the in-memory cache to disk during a restart, whereas python carbon-cache would just lose whatever metrics are in the cache during a process restart.

go-carbon CPU Tuning

- max-cpu - this sets GOMAXPROCS. Use number of cores
- workers - persister worker threads. Use number of cores

There are a bunch more tuning variables required to get go-carbon working well.

These two settings configure how much CPU resources can be used - obviously, having more persister workers is important for throughput.

go-carbon cache Tuning

- max-size - set high, we use 10million
- write-strategy - set to noop

And it’s important to have a big enough cache size to handle any increases in the rate of incoming metrics, or slow downs in the rate of metrics being written. The number is dependant on the resolution of your data and how many metrics you receive in that period. In our current case, we are receiving about 1.5million metrics a minute, and most of the metrics are 60 second resolution so 10 million is a high enough cache size for now. During normal operation the cache size sits about 1.5 - 2 million.

write-strategy defines what order metrics are persisted to disk from the cache. It’s also possible to choose other strategies - either the oldest point first, or the metric with the most unwritten points. noop means “unspecified order” which requires the least CPU and improves cache responsiveness and performance.

go-carbon I/O Tuning

- sysctl settings:
  - `sysctl -w vm.dirty_ratio = 80`
  - `sysctl -w vm.dirty_background_ratio = 50`
  - `sysctl -w vm.dirty_expire_centisecs = $(( 10*60*100 ))`
- max-updates-per-second - we use 10k

go-carbon doesn't try and be too smart about writes - but you need to tune the OS so that go-carbon is never blocked on writes. So we configure these sysctl settings, which come directly from the project README, to tune the kernel disk buffers. The kernel will then take multiple datapoint writes, and coalesce them into a single disk write performed in the background.

dirty_ratio - percentage of RAM which can be left unwritten to disk
dirty_background_ratio - percentage of RAM when the background writer starts writes to disk
dirty_expire_centisecs - this means that memory pages shouldn't left dirty for longer than 10mins

max-updates-per-second is the most important configuration setting to optimise performance. It's name is confusing, at first I thought it is a limit on throughput. in fact, once the amount of datapoints needing to be written goes above the setting value limit, go-carbon will start writing out multiple datapoints per update. so this provides control to limit the amount of IOPs the server needs to perform, and means increasing the cache as datapoints need to be stored in the cache for longer so that go-carbon can write out multiple datapoints for each metric per update.

go-carbon I/O Tuning

- max-creates-per-second - new in 0.13

A really useful feature that python carbon supports is max creates per minute, and happily go-carbon started supporting it in 0.13.

The first time a metric is seen, then carbon will lay out a new whisper file. This means writing out the entire content of the file, which is a lot more IO than just updating a few datapoints. This absolutely kills the update performance of the carbon persister and can affect search performance. So, this setting will rate limit the number of new whisper files being created.

I've not yet updated to 0.13, we're running 0.11, but on python graphite we set this to 1000 per minute, so I'd use a similar value scaled down to seconds here.

go-carbon resourcing

- Enough cores - 2x Xeon E5-2630 2.40GHz maxed out at 3million metrics/minute in my tests
- Ram to support disk buffers - was 24GB, maxed out at ~1.2million metrics/minute, now 48GB
- Need fast SSD

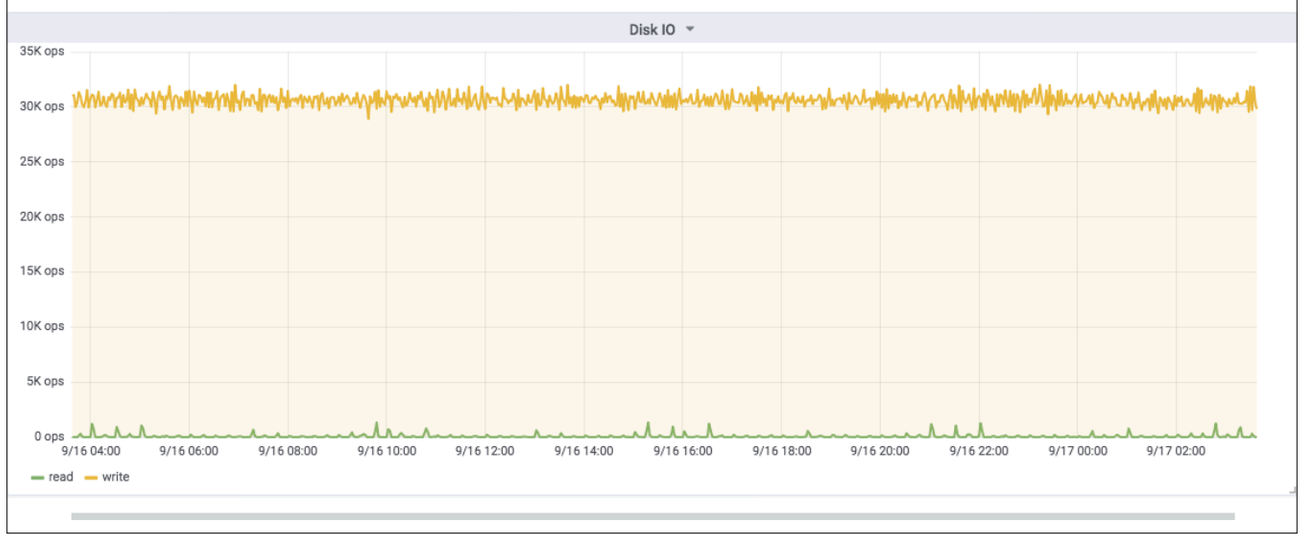
As for system resources, you need enough CPU to manage your workload, I found I couldn't scale beyond 3m/min with 2 CPUs in the lab, so we run with more cores in production to give more headroom.

But the most important resource is I/O throughput. Obviously you need fast enough disks on the backend, but having a large disk buffer in RAM is vital for the operation of go-carbon.

This is because each write to a whisper file requires multiple reads. Also if your whisper files have multiple retention periods, the aggregated points are written to every write, requiring a read of all points in the highest precision that cover that aggregated period. Basically, more cache == less disk reads.

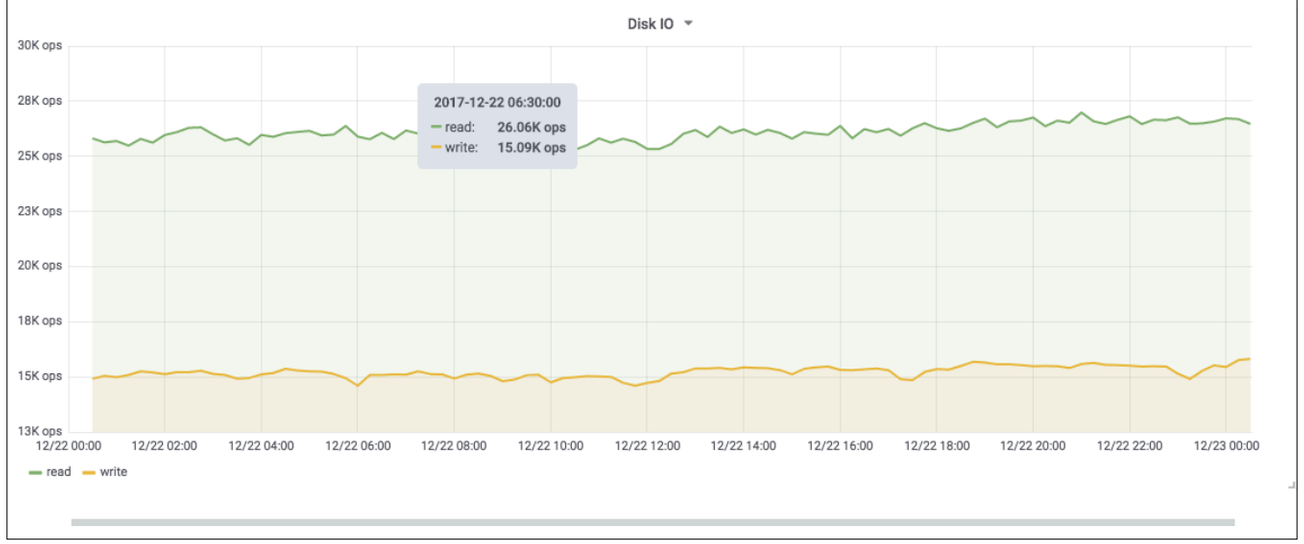
So there's a relationship between the max-writes-per-second setting, the size of your in-memory cache, as well as having a large enough kernel disk cache, that provide the tunings required to get the maximum performance for your hardware from go-carbon.

Good I/O Pattern



So specifically on the I/O and resource tuning - this is the I/O pattern you’re looking for - pretty much entirely writes, with quite a low read rate. This graph shows iops/s on our whisper storage.

Problematic I/O

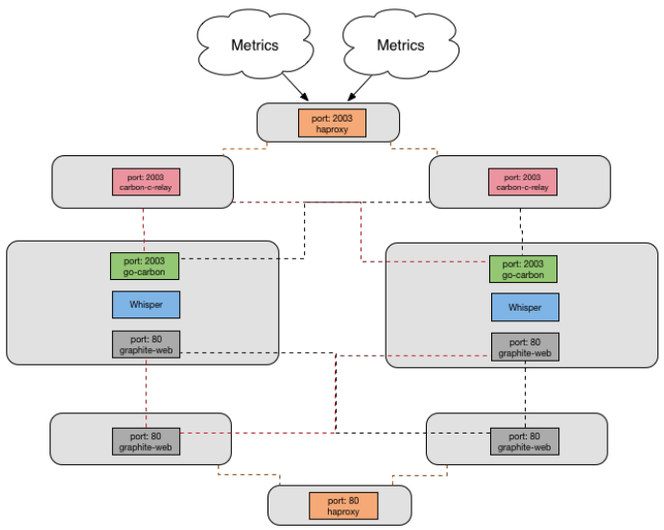


And this is a terrible situation - so many read operations, and writes are struggling. In this scenario you're likely to see your web queries return very slowly, or timeout altogether.

This problem specifically was the system RAM being set too low, and was resolved by increasing the amount of RAM.

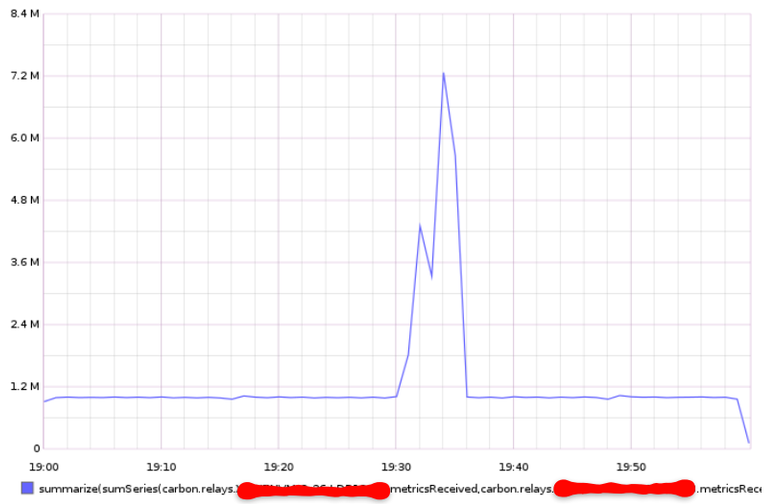
But in general, when you're testing the system or debugging performance problems, check the IO of the disk your whisper files are on.

go-carbon



Now we have our current architecture - go-carbon has replaced all the carbon components, as well as haproxy, on the cache nodes. We still have our node sprawl to support redundancy, but the configuration on each node is much

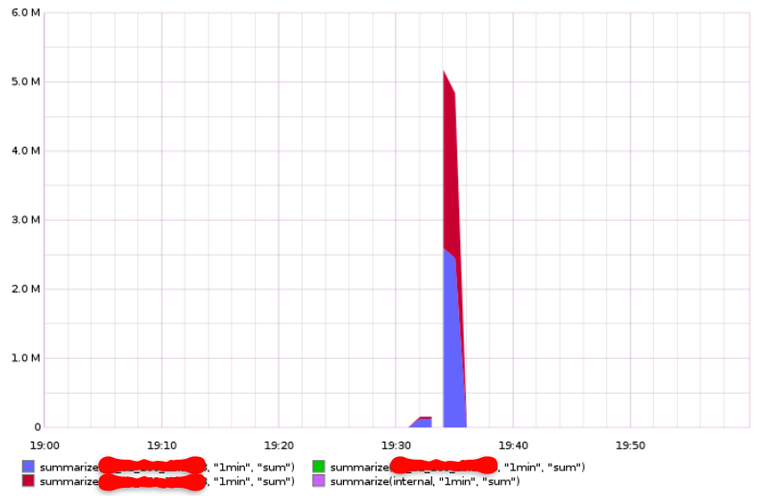
short term metrics burst



So while I was testing go-carbon with out production workload we had an interesting experience. I added a go-carbon node in addition to our python cache nodes. And during this time we had this short spike of incoming metrics.

This graph represents the total all metrics being received by the relay servers, per minute. At the time had a steady ingest rate of around 1million metrics per minute, which is about 16.6k/second. But we have this strange spike, there we burst to 7.2 million, or 120k/sec

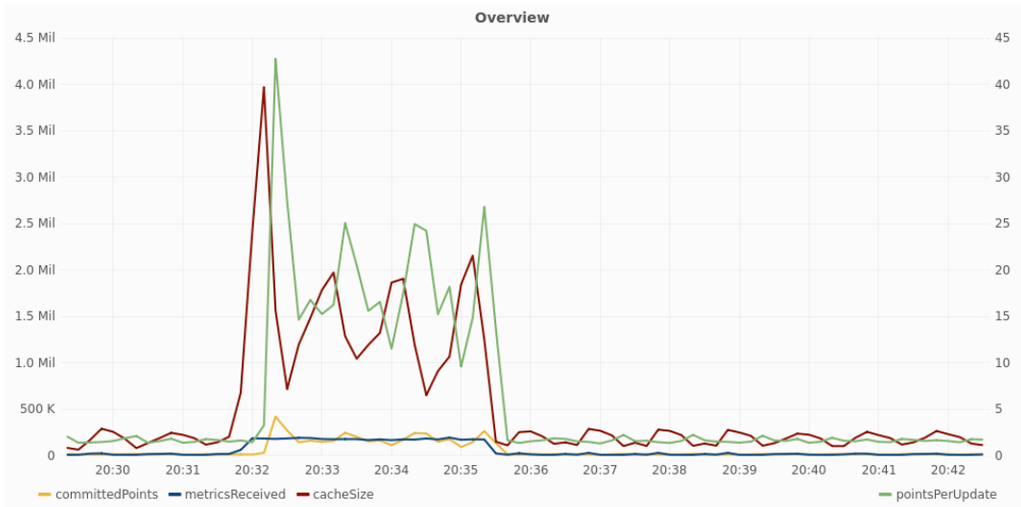
python cache servers drop metrics



This graph shows the number of metrics being dropped by the relay, as the receiving server can’t keep up with the incoming rate of metrics.

The blue and red areas are the python cache servers, and you may be able to see in the legend there’s a green bar as well - that’s our go-carbon server, and it’s not dropping any metrics.

go-carbon server absorbs burst



These are some numbers from the go-carbon cache server in question. Some of the numbers on the left Y scale here are different, as they're being shown per-second in this graph.

The blue line shows what we are receiving from the relay, the yellow line is how many are being persisted, the red line is the size of the internal cache. The green line shows how many datapoints are being written per update, the value is on the right Y axis.

So we saw this spike which is 4-5x our normal ingest rate, and the server was able to take it in it's stride, which was very pleasing to see.

Replacing graphite web

- Composer not much used, mostly just a datasource for grafana
- carbonapi - go replacement with API functionality from graphite-web.
- communicates directly with go-carbon, no web component required on cache servers

After working on the ingest components, we also looked at the replacements for the web and API components.

Specifically we looked at carbonapi, but had problems with it segfaulting. As well as that some functions from graphite weren't available - including groupByNode which I use a lot.

so I decided to stick with graphite web - we didn't have a performance problem from this side, we didn't want to lose the functions we were using and the newer ones that have been coming out, and the information available about carbonapi and debugging these sorts of problems was limited so I didn't want to invest a lot of time in that. But I know large graphite users do use carbonapi in their stack, such as booking.com, so it must work

graphite 1.0 - new functions

- aggregateLine
- applyByNode
- averageOutsidePercentile
- delay
- exponentialMovingAverage
- fallbackSeries
- grep
- groupByNodes
- integralByInterval
- interpolate
- invert
- isNonNull
- linearRegression
- linearRegressionAnalysis
- mapSeries
- movingMin
- movingMax
- movingSum
- multiplySeriesWithWildcards
- offsetToZero
- pow
- powSeries
- reduceSeries
- removeBetweenPercentile
- removeEmptySeries
- sortByTotal
- squareRoot
- timeSlice
- verticalLine
- weightedAverage

Finally, graphite went 1.0 last year, which has introduced a bunch of new functionality.

So there’s been a large batch of new functions added - some of these are really useful, like moving min and max, exponential moving average, and some linear regression functions.

graphite 1.1

- Custom Functions
- Pipe chaining syntax
  - `aliasByNode(movingAverage(sortByName(test.*),”5min”),1)`
  - `test.*|sortByName()|movingAverage("5min")|aliasByNode(1)`

Then graphite 1.1 was released in late 2017, and this included support for custom functions. Quite helpful

Pipe chaining - previous format nested brackets. Now can use pipes, which looks more like the kind of queries you build in grafana. You may prefer this format, I’m pretty used to the brackets

tag support

- Send Metrics

```
my.series;tag1=value1;tag2=value2
```

- Query Metrics

```
seriesByTag('tag1=value1')
```

lastly tag support, which is a bit of a belated effort to offer support for the SQLish search queries supported by the newer TSDBs like influxdb and prometheus.

so this was introduced in python carbon and graphite web first, but go-carbon also added support for tags in 0.12