Graphite Documentation

Release 0.9.9

Chris Davis

May 11, 2012

CONTENTS

CHAPTER

ONE

5-MINUTE OVERVIEW

testing... 1.0 docs

1.1 What Graphite is and is not

Graphite does two things:

1. Store numeric time-series data
2. Render graphs of this data on demand

What Graphite does not do is collect data for you, however there are some *tools* out there that know how to send data to graphite. Even though it often requires a little code, *sending data* to Graphite is very simple.

1.2 About the project

Graphite is an enterprise-scale monitoring tool that runs well on cheap hardware. It was originally designed and written by Chris Davis at Orbitz in 2006 as side project that ultimately grew to be a foundational monitoring tool. In 2008, Orbitz allowed Graphite to be released under the open source Apache 2.0 license. Since then Chris has continued to work on Graphite and has deployed it at other companies including Sears, where it serves as a pillar of the e-commerce monitoring system. Today many large *companies* use it.

1.3 The architecture in a nutshell

Graphite consists of 3 software components:

1. **carbon** - a Twisted daemon that listens for time-series data
2. **whisper** - a simple database library for storing time-series data (similar in design to RRD)
3. **graphite webapp** - A Django webapp that renders graphs on-demand using Cairo

Feeding in your data is pretty easy, typically most of the effort is in collecting the data to begin with. As you send datapoints to Carbon, they become immediately available for graphing in the webapp. The webapp offers several ways to create and display graphs including a simple *URL API* that makes it easy to embed graphs in other webpages.

CHAPTER

TWO

INSTALLING GRAPHITE

2.1 Dependencies

Since Graphite renders graphs using Cairo, it depends on several graphics-related libraries not typically found on a server. If you're installing from source you can use the `check-dependencies.py` script to see if the dependencies have been met or not.

In general, Graphite requires:

- Python 2.4 or greater (2.6+ recommended)
- Pycairo
- Django 1.0 or greater
- django-tagging 0.3.1
- A json module, if you're using Python2.6 this comes standard. With 2.4 you should install simplejson
- A Django-supported database module (sqlite comes standard with Python 2.6)
- Twisted 8.0 or greater (10.0+ recommended)

Also both the Graphite webapp and Carbon require the whisper database library.

There are also several optional dependencies, some of which are necessary for high performance.

- Apache with mod_wsgi or mod_python (mod_wsgi preferred)
- memcached and python-memcache
- python-ldap (for LDAP authentication support in the webapp)
- txamqp (for AMQP support in Carbon)

See Also:

On some systems it is necessary to install some fonts, if you get the webapp running and only see broken images instead of graphs, this is probably why.

- https://answers.launchpad.net/graphite/+question/38833
- https://answers.launchpad.net/graphite/+question/133390
- https://answers.launchpad.net/graphite/+question/127623

2.2 Tips For Fulfilling Dependencies

Usually the hardest dependency to fulfill is Pycairo because it requires Cairo, which in turn requires fontconfig, etc... Often your distribution's package manager will be able to install cairo and all of its dependencies for you, but in order to build Pycairo (which is often *not* covered by the package manager) you'll need the *cairo-devel* package so C headers are available.

2.3 Binary Packages

We are currently working on getting RPMs and DEB packages ready for Graphite. As of this writing, Whisper is available in Ubuntu. To install it you can simply:

```
apt-get install python-whisper
```

The Graphite webapp and Carbon do not yet have binary packages available.

2.4 Installing From Source

You can download the latest source tarballs for grahite, carbon, and whisper from the Graphite project page, https://launchpad.net/graphite

To install, simply extract the tarball and install like any other python package.

```
# First we install whisper, as both Carbon and Graphite require it
tar zxf whisper-0.9.8.tgz
cd whisper-0.9.8/
sudo python2.6 setup.py install
cd ..
# Now we install carbon
tar zxf carbon-0.9.8.tgz
cd carbon-0.9.8/
sudo python2.6 setup.py install
cd ..
# Finally, the graphite webapp
tar zxf graphite-web-0.9.8.tgz
cd graphite-web-0.9.8/
./check-dependencies.py
# once all dependencies are met...
sudo python2.6 setup.py install
```

This will install whisper as a site-package, while Carbon and Graphite will be installed in `/opt/graphite/`.

2.5 Help! It didn't work!

If you run into any issues with Graphite, feel free to post a question to our Questions forum on Launchpad

2.6 Post-Install Tasks

Configuring Carbon Once you've installed everything you will need to create some basic configuration. Initially none of the config files are created by the installer but example files are provided. Simply copy the `.example`

files and customize.

`Administering Carbon` Once Carbon is configured, you need to start it up.

Feeding In Your Data Once it's up and running, you need to feed it some data.

Configuring The Webapp With data getting into carbon, you probably want to look at graphs of it. So now we turn our attention to the webapp.

Administering The Webapp Once its configured you'll need to get it running.

Using the Composer Now that the webapp is running, you probably want to learn how to use it.

That covers the basics, the next thing you should probably read about is *The URL API*.

CHAPTER

THREE

THE CARBON DAEMONS

When we talk about "Carbon" we mean one or more of various daemons that make up the storage backend of a Graphite installation. In simple installations, there is typically only one daemon, `carbon-cache.py`. This document gives a brief overview of what each daemon does and how you can use them to build a more sophisticated storage backend.

All of the carbon daemons listen for time-series data and can accept it over a common set of *protocols*. However, they differ in what they do with the data once they receive it.

3.1 carbon-cache.py

`carbon-cache.py` accepts metrics over various protocols and writes them to disk as efficiently as possible. This requires caching metric values in RAM as they are received, and flushing them to disk on an interval using the underlying *whisper* library.

`carbon-cache.py` requires some basic configuration files to run:

carbon.conf The `[cache]` section tells `carbon-cache.py` what ports (2003/2004/7002), protocols (newline delimited, pickle) and transports (TCP/UDP) to listen on.

storage-schemas.conf Defines a retention policy for incoming merics based on regex patterns. This policy is passed to *whipser* when the `.wsp` file is pre-allocated, and dictates how long data is stored for.

As the number of incoming metrics increases, one `carbon-cache.py` instance may not be enough to handle the I/O load. To scale out, simply run multiple `carbon-cache.py` instances (on one or more machines) behind a `carbon-aggregator.py` or `carbon-relay.py`.

3.2 carbon-relay.py

`carbon-relay.py` serves two distinct purposes: replication and sharding.

When running with `RELAY_METHOD = rules`, a `carbon-relay.py` instance can run in place of a `carbon-cache.py` server and relay all incoming metrics to multiple backend `carbon-cache.py`'s running on different ports or hosts.

In `RELAY_METHOD = consistent-hashing` mode, a `CH_HOST_LIST` setting defines a sharding strategy across multiple `carbon-cache.py` backends. The same consistent hashing list can be provided to the graphite webapp via `CARBONLINK_HOSTS` to spread reads across the multiple backends.

`carbon-relay.py` is configured via:

carbon.conf The `[relay]` section defines listener host/ports and a `RELAY_METHOD`

relay-rules.conf In `RELAY_METHOD = rules`, pattern/servers tuples define what servers metrics matching certain regex rules are forwarded to.

3.3 carbon-aggregator.py

`carbon-aggregator.py` can be run in front of `carbon-cache.py` to buffer metrics over time before reporting them into *whisper*. This is useful when granular reporting is not required, and can help reduce I/O load and whisper file sizes due to lower retention policies.

`carbon-aggregator.py` is configured via:

carbon.conf The `[aggregator]` section defines listener and destination host/ports.

aggregation-rules.conf Defines a time interval (in seconds) and aggregation function (sum or average) for incoming metrics matching a certain pattern. At the end of each interval, the values received are aggregated and published to `carbon-cache.py` as a single metric.

CHAPTER

FOUR

CONFIGURING CARBON

Carbon's config files all live in `/opt/graphite/conf/`. If you've just installed Graphite, none of the `.conf` files will exist yet, but there will be a `.conf.example` file for each one. Simply copy the example files, removing the .example extension, and customize your settings.

4.1 carbon.conf

This is the main config file defines the settings for each Carbon daemon. If this is your first time using Graphite, don't worry about anything but the `[cache]` section for now. If you're curious you can read about *The Carbon Daemons*.

4.2 storage-schemas.conf

This file defines how much data to store, and at what precision. Important notes before continuing:

- There can be many sections in this file.
- Each section must have a header in square brackets, a pattern and a retentions line.
- The sections are applied in order from the top (first) and bottom (last).
- The patterns are regular expressions, as opposed to the wildcards used in the URL API.
- The first pattern that matches the metric name is used.
- These are set at the time the first metric is sent.
- Changing this file will not affect .wsp files already created on disk. Use whisper-resize.py to change those.
- There are two very different ways to specify retentions. We will show the new, easier way first, and the old, more difficult way second for historical purposes second.

Here's an example:

```
[garbage_collection]
pattern = garbageCollections$
retentions = 10s:14d
```

The name [garbage_collection] is only used so that you know what section this is handling, and so that the parser knows a new section has started.

The pattern above will match any metric that ends with `garbageCollections`. For example, `com.acmeCorp.javaBatch01.instance01.jvm.memory.garbageCollections` would match, but `com.acmeCorp.javaBatch01.instance01.jvm.memory.garbageCollections.full` would not.

The retention line is saying that each 'slot' or 'datapoint' represents 10 seconds, and we want to keep enough slots so that they add up to 14 days of data.

Here's a more complicated example with multiple retention rates:

```
[apache_busyWorkers]
pattern = servers\.www.*\.workers\.busyWorkers$
retentions = 15s:7d,1m:21d,15m:5y
```

The pattern matches server names that start with 'www', followed by anything, that end in '.workers.busyWorkers'. This way not all metrics associated with your webservers need this type of retention.

As you can see there are multiple retentions. Each is used in the order that it is provided. As a general rule, they should be in most-precise:shortest-length to least-precise:longest-time. Retentions are merely a way to save you disk space and decrease I/O for graphs that span a long period of time. By default, when data moves from a higher precision to a lower precision, it is **averaged**. This way, you can still find the **total** for a particular time period if you know the original precision. (To change the aggregation method, see the next section.)

Example: You store the number of sales per minute for 1 year, and the sales per hour for 5 years after that. You need to know the total sales for January 1st of the year before. You can query whisper for the raw data, and you'll get 24 datapoints, one for each hour. They will most likely be floating point numbers. You can take each datapoint, multiply by 60 (the ratio of high-precision to low-precision datapoints) and still get the total sales per hour.

The old retentions was done as follows:

```
retentions = 60:1440
```

60 represents the number of seconds per datapoint, and 1440 represents the number of datapoints to store. This required some unnecessarily complicated math, so although it's valid, it's not recommended. It's left in so that large organizations with complicated retention rates need not re-write their storage-schemas.conf while when they upgrade.

4.3 storage-aggregation.conf

This file defines how to aggregate data to lower-precision retentions. The format is similar to `storage-schemas.conf`. Important notes before continuing:

- This file is optional. If it is not present, defaults will be used.
- There is no `retentions` line. Instead, there are `xFilesFactor` and/or `aggregationMethod` lines.
- `xFilesFactor` should be a floating point number between 0 and 1, and specifies what fraction of the previous retention level's slots must have non-null values in order to aggregate to a non-null value. The default is 0.5.
- `aggregationMethod` specifies the function used to aggregate values for the next retention level. Legal methods are `average`, `sum`, `min`, `max`, and `last`. The default is `average`.
- These are set at the time the first metric is sent.
- Changing this file will not affect .wsp files already created on disk. Use whisper-resize.py to change those.

Here's an example:

```
[all_min]
pattern = \.min$
xFilesFactor = 0.1
aggregationMethod = min
```

The pattern above will match any metric that ends with `.min`.

The `xFilesFactor` line is saying that a minimum of 10% of the slots in the previous retention level must have values for next retention level to contain an aggregate. The `aggregationMethod` line is saying that the aggregate function to use is `min`.

If either `xFilesFactor` or `aggregationMethod` is left out, the default value will be used.

The aggregation parameters are kept separate from the retention parameters because the former depends on the type of data being collected and the latter depends on volume and importance.

4.4 relay-rules.conf

Relay rules are used to send certain metrics to a certain backend. This is handled by the carbon-relay system. It must be running for relaying to work. You can use a regular expression to select the metrics and define the servers to which they should go with the servers line.

Example:

```
[example]
pattern = ^mydata\.foo\..+
servers = 10.1.2.3, 10.1.2.4:2004, myserver.mydomain.com
```

You must define at least one section as the default.

4.5 aggregation-rules.conf

Aggregation rules allow you to add several metrics together as the come in, reducing the need to sum() many metrics in every URL. Note that unlike some other config files, any time this file is modified it will take effect automatically. This requires the carbon-aggregator service to be running.

The form of each line in this file should be as follows:

```
output_template (frequency) = method input_pattern
```

This will capture any received metrics that match 'input_pattern' for calculating an aggregate metric. The calculation will occur every 'frequency' seconds and the 'method' can specify 'sum' or 'avg'. The name of the aggregate metric will be derived from 'output_template' filling in any captured fields from 'input_pattern'.

For example, if you're metric naming scheme is:

```
<env>.applications.<app>.<server>.<metric>
```

You could configure some aggregations like so:

```
<env>.applications.<app>.all.requests (60) = sum <env>.applications.<app>.*.requests
<env>.applications.<app>.all.latency (60) = avg <env>.applications.<app>.*.latency
```

As an example, if the following metrics are received:

```
prod.applications.apache.www01.requests
prod.applications.apache.www02.requests
prod.applications.apache.www03.requests
prod.applications.apache.www04.requests
prod.applications.apache.www05.requests
```

They would all go into the same aggregation buffer and after 60 seconds the aggregate metric 'prod.applications.apache.all.requests' would be calculated by summing their values.

CHAPTER

FIVE

FEEDING IN YOUR DATA

Getting your data into Graphite is very flexible. There are three main methods for sending data to Graphite: Plaintext, Pickle, and AMQP.

It's worth noting that data sent to Graphite is actually sent to the *Carbon and Carbon-Relay*, which then manage the data. The Graphite web interface reads this data back out, either from cache or straight off disk.

Choosing the right transfer method for you is dependent on how you want to build your application or script to send data:

- For a singular script, or for test data, the plaintext protocol is the most straightforward method.
- For sending large amounts of data, you'll want to batch this data up and send it to Carbon's pickle receiver.
- Finally, Carbon can listen to a message bus, via AMQP.

5.1 The plaintext protocol

The plaintext protocol is the most straightforward protocol supported by Carbon.

The data sent must be in the following format: `<metric path> <metric value> <metric timestamp>`. Carbon will then help translate this line of text into a metric that the web interface and Whisper understand.

On Unix, the `nc` program can be used to create a socket and send data to Carbon (by default, 'plaintext' runs on port 2003):

```
PORT=2003
SERVER=graphite.your.org
echo "local.random.diceroll 4 `date +%s`" | nc ${SERVER} ${PORT};
```

5.2 The pickle protocol

The pickle protocol is a much more efficient take on the plaintext protocol, and supports sending batches of metrics to Carbon in one go.

The general idea is that the pickled data forms a list of multi-level tuples:

```
[(path, (timestamp, value)), ...]
```

Once you've formed a list of sufficient size (don't go too big!), send the data over a socket to Carbon's pickle receiver (by default, port 2004). You'll need to pack your pickled data into a packet containing a simple header:

```
payload = pickle.dumps(listOfMetricTuples)
header = struct.pack("!L", len(payload))
message = header + payload
```

You would then send the `message` object through a network socket.

5.3 Using AMQP

...

CHAPTER

SIX

CONFIGURING THE WEBAPP

CHAPTER

SEVEN

ADMINISTERING THE WEBAPP

CHAPTER

EIGHT

USING THE COMPOSER

...

CHAPTER

NINE

THE URL API

The graphite webapp provides a `/render` endpoint for generating graphs (and retreiving raw data). This endpoint accepts various arguments via query string parameters. These parameters are separated by an ampersand (`&`) and are supplied in the format:

```
&name=value
```

To verify that the api is running and able to generate images, open `http://GRAPHITE_HOST:GRAPHITE_PORT/render` in a browser. The api should return a simple 330x250 image with the text "No Data".

Once the api is running and you've begun *feeding data into carbon*, use the parameters below to customize your graphs and pull out raw data. For example:

```
# single server load on large graph
http://graphite/render?target=server.web1.load&height=800&width=600

# average load across web machines over last 12 hours
http://graphite/render?target=averageSeries(server.web*.load)&from=-12hours

# number of registered users over past day as raw json data
http://graphite/render?target=app.numUsers&format=json

# rate of new signups per minute
http://graphite/render?target=summarize(deriviative(app.numUsers),"1min")&title=New_Users_Per_Minute
```

Note: Most of the functions and parameters are case sensitive. For example `&linewidth=2` will fail silently. The correct parameter is `&lineWidth=2`

9.1 Graphing Metrics

To begin graphing specific metrics, pass one or more target parameters and specify a time window for the graph via from / until.

9.1.1 target

This will draw one or more metrics

Example:

```
&target=company.server05.applicationInstance04.requestsHandled
(draws one metric)
```

Let's say there are 4 identical application instances running on each server.

```
&target=company.server05.applicationInstance*.requestsHandled
(draws 4 metrics / lines)
```

Now let's say you have 10 servers.

```
&target=company.server*.applicationInstance*.requestsHandled
(draws 40 metrics / lines)
```

You can also run any number of *functions* on the various metrics before graphing.

```
&target=averageSeries(company.server*.applicationInstance.requestsHandled)
(draws 1 aggregate line)
```

The target param can also be repeated to graph multiple related metrics.

```
&target=company.server1.loadAvg&target=company.server1.memUsage
```

Note: If more than 10 metrics are drawn the legend is no longer displayed. See the hideLegend parameter for details.

9.1.2 from / until

These are optional parameters that specify the relative or absolute time period to graph. `&from` specifies the beginning, `&until` specifies the end. If `&from` is omitted, it defaults to 24 hours ago. If `&until` is omittied, it defaults to the current time (now).

There are multiple formats for these functions:

```
&from=-RELATIVE_TIME
&from=ABSOLUTE_TIME
```

RELATIVE_TIME is a length of time since the current time. It is always preceded my a minus sign (-) and follow by a unit of time. Valid units of time:

Abbrieviation	Unit
s	Seconds
min	Minutes
h	Hours
d	Days
w	Weeks
mon	30 Days (month)
y	365 Days (year)

ABSOLUTE_TIME is in the format HH:MM_YYMMDD, YYYYMMDD, MM/DD/YY, or any other `at(1)`-compatible time format.

Abbreiviation	Meaning
HH	Hours, in 24h clock format. Times before 12PM must include leading zeroes.
MM	Minutes
YYYY	4 Digit Year.
MM	Numeric month representation with leading zero
DD	Day of month with leadng zero

`&from` and `&until` can mix absolute and relative time if desired.

Examples:

```
&from=-8d&until=-7d
(shows same day last week)

&from=04:00_20110501&until=16:00_20110501
(shows 4AM-4PM on May 1st, 2011)

&from=20091201&until=20091231
(shows December 2009)

&from=noon+yesterday
(shows data since 12:00pm on the previous day)

&from=6pm+today
(shows data since 6:00pm on the same day)

&from=january+1
(shows data since the beginning of the current year)

&from=monday
(show data since the previous monday)
```

9.2 Retreving Data

Instead of rendering an image, the api can also return the raw data in various formats for external graphing, analysis or monitoring.

9.2.1 rawData

Note: This option is deprecated in favor of `&format`

Used to get numerical data out of the webapp instead of an image. Can be set to true, false, csv. Affects all `&targets` passed in the URL.

Example:

```
&target=carbon.agents.graphiteServer01.cpuUsage&from=-5min&rawData=true
```

Returns the following text:

```
carbon.agents.graphiteServer01.cpuUsage,1306217160,1306217460,60|0.0,0.00666666520965,0.006666666242
```

9.2.2 format

Returns raw data instead of a graph. Affects all `&targets` passed in the URL.

Examples:

```
&format=raw
&format=csv
&format=json
```

raw

```
entries,1311836008,1311836013,1|1.0,2.0,3.0,5.0,6.0
```

csv

```
entries,2011-07-28 01:53:28,1.0
entries,2011-07-28 01:53:29,2.0
entries,2011-07-28 01:53:30,3.0
entries,2011-07-28 01:53:31,5.0
entries,2011-07-28 01:53:32,6.0
```

json

```
[{
  "target": "entries",
  "datapoints": [
    [1.0, 1311836008],
    [2.0, 1311836009],
    [3.0, 1311836010],
    [5.0, 1311836011],
    [6.0, 1311836012]
  ]
}]
```

9.3 Customizing Graphs

9.3.1 width / height

```
&width=XXX&height=XXX
```

These are optional parameters that define the image size in pixels

Example:

```
&width=650&height=250
```

9.3.2 template

Used to specify a template from `graphTemplates.conf` to use for default colors and graph styles.

Example:

```
&template=plain
```

9.3.3 margin

Used to increase the margin around a graph image on all sides. Must be passed a positive integer. If omitted, the default margin is 10 pixels.

Example:

```
&margin=20
```

9.3.4 bgcolor

Sets the background color of the graph.

Color Names	RGB Value
black	0,0,0
white	255,255,255
blue	100,100,255
green	0,200,0
red	200,0,50
yellow	255,255,0
orange	255, 165, 0
purple	200,100,255
brown	150,100,50
aqua	0,150,150
gray	175,175,175
grey	175,175,175
magenta	255,0,255
pink	255,100,100
gold	200,200,0
rose	200,150,200
darkblue	0,0,255
darkgreen	0,255,0
darkred	255,0,0
darkgray	111,111,111
darkgrey	111,111,111

RGB can be passed directly in the format #RRGGBB where RR, GG, and BB are 2-digit hex vaules for red, green and blue, respectively.

Examples:

```
&bgcolor=blue
&bgcolor=#2222FF
```

9.3.5 fgcolor

Sets the foreground color. This only affects the title, legend text, and axis labels.

See majorGridLineColor, and minorGridLineColor to change more of the graph to your preference.

See bgcolor for a list of color names and details on formatting this parameter.

9.3.6 fontName

Change the font used to render text on the graph. The font must be installed on the Graphite Server.

Example:

```
&fontName=FreeMono
```

9.3.7 fontSize

Changes the font size. Must be passed a positive floating point number or integer equal to or greater than 1. Default is 10

Example:

```
&fontSize=8
```

9.3.8 fontBold

If set to true, makes the font bold. Default is false.

Example:

```
&fontBold=true
```

9.3.9 fontItalic

If set to true, makes the font italic / oblique. Default is false.

Example:

```
&fontItalic=true
```

9.3.10 yMin

Manually sets the lower bound of the graph. Can be passed any integer or floating point number. By deafult, Graphite attempts to fit all data on one graph.

Example:

```
&yMin=0
```

9.3.11 yMax

Manually sets the upper bound of the graph. Can be passed any integer or floating point number. By deafult, Graphite attempts to fit all data on one graph.

Example:

```
&yMax=0.2345
```

9.3.12 colorList

Passed one or more comma-separated color names or RGB values (see bgcolor for a list of color names) and uses that list in order as the colors of the lines. If more lines / metrics are drawn than colors passed, the list is reused in order.

Example:

```
&colorList=green,yellow,orange,red,purple,#DECAFF
```

9.3.13 title

Puts a title at the top of the graph, center aligned. If omitted, no title is displayed.

Example:

```
&title=Apache Busy Threads, All Servers, Past 24h
```

9.3.14 vtitle

Labels the y-axis with vertical text. If omitted, no y-axis labe is displayed.

Example:

```
&vtitle=Threads
```

9.3.15 lineMode

Sets the type of line to be drawn. Valid modes are 'staircase' (each data point is flat for the duration of the time period) and 'slope' (comes to a point at the time, and slopes to the next time.) If omitted, default is 'slope'.

Example:

```
&lineMode=staircase
```

9.3.16 lineWidth

Takes any floating point or integer. (negative numbers do not error but will cause no line to be drawn. Changes the width of the line in pixels.

Example:

```
&lineWidth=2
```

9.3.17 hideLegend

If set to 'true', the legend is not drawn. If set to 'false', the legend is drawn.

Default value changes depending on the number of targets. If there are 10 or less targets, default is true. If there are more than 10 targets, default is false.

You can force the legend to be draw for more than 10 targets by setting this to false. You may need to increase the `&height` parameter to accomodate the additional text.

Example:

```
&hideLegend=false
```

9.3.18 hideAxes

true or false. Hides the x- and y-axes. Default is false.

Example:

```
&hideAxes=true
```

9.3.19 hideGrid

true or false Hides the grid lines. Default is false.

Example:

```
&hideGrid=true
```

9.3.20 minXStep

9.3.21 majorGridLineColor

Sets the color of the major grid lines.

See bgcolor for valid color names and formats.

Example:

```
&majorGridLineColor=#FF22FF
```

9.3.22 minorGridLineColor

Sets the color of the minor grid lines.

See bgcolor for valid color names and formats.

Example:

```
&minorGridLineColor=darkgrey
```

9.3.23 thickness

Alias for lineWidth

9.3.24 min

alias for yMin

9.3.25 max

alias for yMax

9.3.26 tz

Time zone to convert all times into.

Examples:

```
&tz=America/Los_Angeles
&tz=UTC
```

Note: To change the default timezone, edit `webapp/graphite/local_settings.py`.

CHAPTER

TEN

FUNCTIONS

CHAPTER

ELEVEN

THE DASHBOARD UI

...

CHAPTER

TWELVE

TOOLS THAT WORK WITH GRAPHITE

12.1 Diamond

Diamond is a Python daemon that collects system metrics and publishes them to Graphite. It is capable of collecting cpu, memory, network, I/O, load and disk metrics. Additionally, it features an API for implementing custom collectors for gathering metrics from almost any source.

12.2 jmxtrans

jmxtrans is a powerful tool that performs JMX queries to collect metrics from Java applications. It is requires very little configuration and is capable of sending metric data to several backend applications, including Graphite.

12.3 statsd

statsd is a simple daemon for easy stats aggregation, developed by the folks at Etsy.

12.4 Ganglia

Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It collects system performance metrics and stores them in RRD, but now there is an add-on that allows Ganglia to send metrics directly to Graphite. Further integration work is underway.

12.5 collectd

collectd is a daemon which collects system performance statistics periodically and provides mechanisms to store the values in a variety of ways, including RRD. To send collectd merics into carbon/graphite, use:

- Jordan Sissel's node collectd-to-graphite proxy
- Joe Miller's perl collectd-graphite plugin
- Gregory Szorc's python collectd-carbon plugin
- Scott Sanders's C collectd-write_graphite plugin

Graphite can also read directly from collectd's RRD files. RRD files can simply be added to `STORAGE_DIR/rrd` (as long as directory names and files do not contain any `.` characters). For example, collectd's `host.name/load/load.rrd` can be symlinked to `rrd/collectd/host_name/load/load.rrd` to graph `collectd.host_name.load.load.{short,mid,long}term`.

12.6 Logster

Logster is a utility for reading log files and generating metrics in Graphite or Ganglia. It is ideal for visualizing trends of events that are occurring in your application/system/error logs. For example, you might use logster to graph the number of occurrences of HTTP response code that appears in your web server logs.

12.7 Rocksteady

A system that ties together Graphite, RabbitMQ, and Esper. Developed by AdMob (who was then bought by Google), this was released by Google as open source (http://google-opensource.blogspot.com/2010/09/get-ready-to-rocksteady.html). Learn more here, http://code.google.com/p/rocksteady/

CHAPTER

THIRTEEN

WHO IS USING GRAPHITE?

Here are some organizations that use Graphite:

- Orbitz
- Sears Holdings
- Etsy (see http://codeascraft.etsy.com/2010/12/08/track-every-release/)
- Google (opensource Rocksteady project)
- Media Temple
- Canonical

And many more, I still need to collect some links...

CHAPTER

FOURTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*