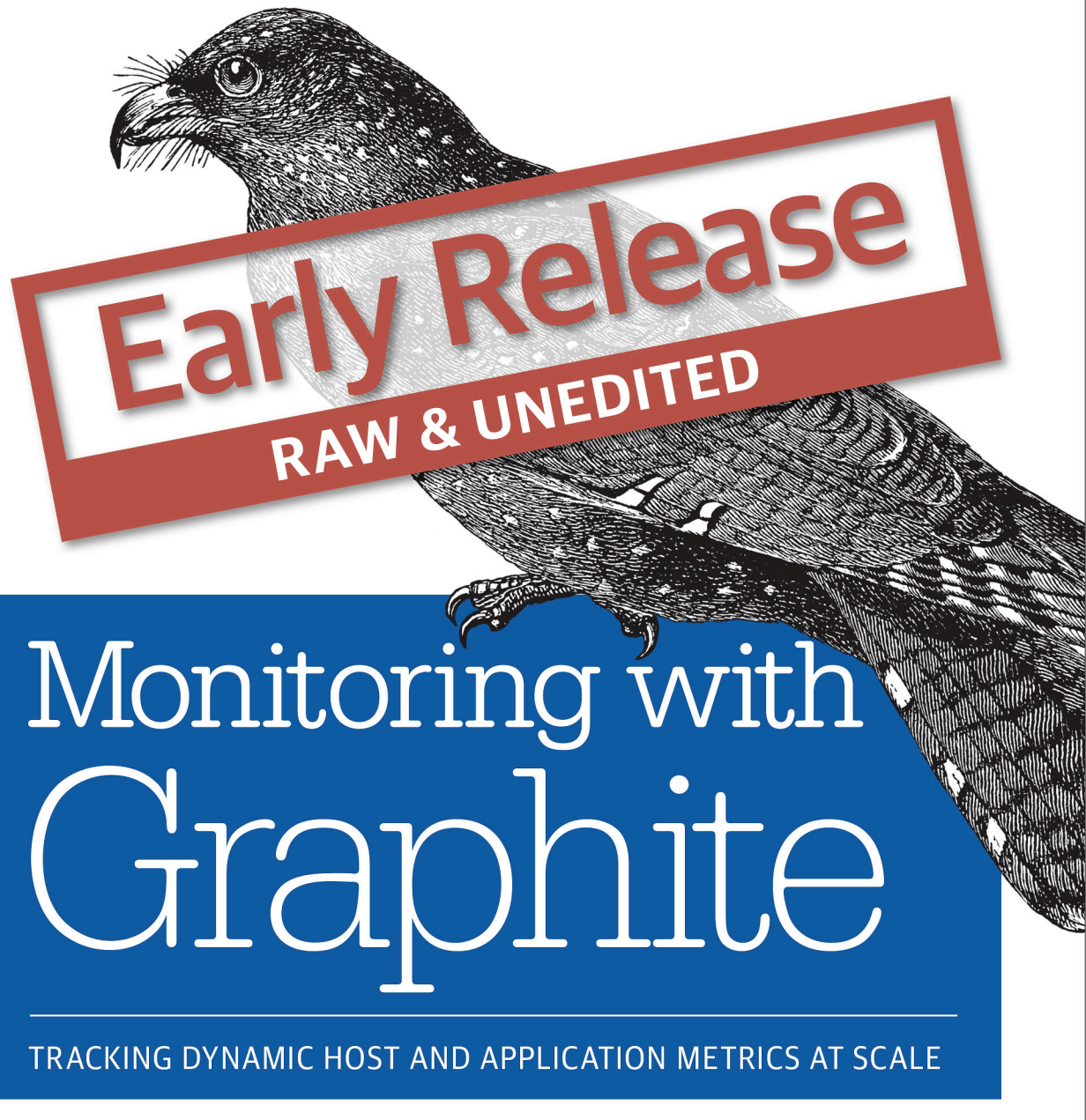
O'REILLY®



Monitoring with Graphite

TRACKING DYNAMIC HOST AND APPLICATION METRICS AT SCALE

Jason Dixon

Monitoring with Graphite

Jason Dixon

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Monitoring with Graphite

by Jason Dixon

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Anderson
Production Editor: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!

Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition:

2014-12-16: Early release revision 1

2015-02-09: Early release revision 2

2015-03-27: Early release revision 3

2015-05-27: Early release revision 4

See *http://oreilly.com/catalog/errata.csp?isbn=0636920035794* for release details.

ISBN: 063-6-920-03579-4

[?]

Table of Contents

Preface

This book is a work in progress – new chapters will be added as they are written. We welcome feedback – if you spot any errors or would like to suggest improvements, please email the author at *graphitebook@obfuscurity.com*.

Twenty years ago it was considered *bleeding-edge* to monitor your Internet router with SNMP and a Perl script. This gave network administrators a new level of insight into their operations and user activity. As the popularity of the Internet soared, so too did the number of successful online businesses and services. This, predictably, led to better monitoring and trending software. On the one hand, businesses had to defend their investment against service and network outages; on the other, they needed improved methods for capacity planning as their systems scaled to meet user demand.

Graphite is one of the most powerful monitoring tools available today. Its popularity is driven not only by its *Open Source* availability, but its ease of use, rapid graph prototyping abilities, and a friendly rendering API that allows anyone to embed Graphite charts in their own applications or websites. The Graphite user community is huge and routinely feeding enhancements back into the core project, such as new statistical and transformative functions, output formats like JSON for client-side chart rendering, and even pluggable storage backends to leverage the growing ecosystem of distributed database systems.

Who Should Read This Book

Monitoring with Graphite is for anyone who wants to learn more about monitoring systems, services or applications. The book makes few assumptions about your background or experiences other than that you have access to a computer for setting up and interacting with your own Graphite instance.

Systems Administrators and Users who wish to gain advanced skills related to scaling Graphite should be comfortable with a Linux or UNIX-style command line environment. You won't be dropped in the morass and expected to fend for yourself, but you'll get more out of this experience if you already know how to navigate a UNIX filesystem, interact with package managers, and what to do when disk fills up due to an excess of Whisper files.

Everyone should be prepared to have fun learning about monitoring best practices, time-series data storage and retrieval, and constructing charts with a toolkit full of rendering functions and statistical primitives. You *will* impress your friends and family with the skills you take away from this volume. I almost guarantee it.

Last but certainly not least, understanding how to use and administer Graphite is a valuable skill. Search any online technical job board and you're sure to find a number of opportunities for individuals that know how to use Graphite effectively or better yet, install and administer it.

Why I Wrote This Book

In spite of Graphite's popularity, advanced knowledge around the maintenance and "scaling out" of its software components tends to be concentrated among a handful of experienced users and developers. As a core developer and maintainer of the Graphite project, I've seen the effects of this knowledge gap first-hand as businesses and individuals approach me routinely for scaling advice and demonstrations. Unfortunately, I don't scale nearly as well as the project I love. Therefore, it only makes sense for me to relent and communicate my experiences in this book.

Some of my friends refer to me as "the Graphite Whisperer". As ridiculous as the title may sound, I admit that I feel a certain pride arranging an elaborate sequence of statistical functions into a graph that accurately represents the source data. I hope that I'm able to imbue you, the reader, with as much Graphite mastery as you can endure. Understandably, not everyone gets as excited talking about time-series line charts as I do. I'll do my best to keep the forthcoming content instructional yet entertaining, relevant and applicable to the most common use cases.

A Word on Monitoring Today

Three years ago I was writing about how "#monitoringsucks" and we need to reconsider our approach to monitoring architectures. Traditional monitoring systems were monolithic and unwieldy. Nagios already had a largely negative connotation but there were few alternatives on the horizon.

Either by luck or instinct, I correctly predicted that the monitoring ecosystem would evolve towards a composable model of well-defined services and compatible interfaces.

Look at the Open Source monitoring community and this is precisely what you'll find today.

In particular, the emergence of NoSQL databases has resulted in enormous momentum and competition among the various time-series databases. Users and businesses have embraced Etsy's "Measure Anything, Measure Everything" mantra, which has driven the need for larger and faster data stores. A hugely successful conference series arose out of the desire for more collaboration and discussion around our monitoring toolchain. People are more open and transparent about their processes and tooling and these shared experiences benefit everyone.

In other words, software moves fast and Graphite is no exception. Fortunately, Graphite has reached such a high level of adoption among users and businesses that you'll be hard-pressed to find tools and services that don't integrate with it directly through its robust APIs or among any number of third-party bridge utilities. In many regards, Graphite has become a time-series specification that other software projects leverage for themselves.

Navigating This Book

This book is organized roughly as follows:

- Chapters 1 and 2 provide a basic introduction to monitoring and trending concepts and terminologies. There's a large vocabulary of terms shared among Graphite users; it helps to speak the same language.
- Chapter 3 discusses various approaches to instrumentation and gathering Telemetry data. Engineers in particular should ideally take a lot of new concepts away from reading this chapter.
- Chapters 4 and 5 introduce the components that make up Graphite and their respective features and functionality. You will learn how to install Graphite and configure it for a basic setup.
- Chapter 6 and 7 cover the typical user workflow for creating your first line chart. By the end of these chapters you should be very comfortable building complex charts with chained functions, multiple axes and interacting with the rendering API directly.
- Chapter 8 introduces the native Graphite dashboard and some of the more popular third-party external dashboards. You'll also get a more thorough coverage of the render API and how to leverage it for client-side chart rendering with javascript frameworks like D3.js.

- Chapters 9 through 12 are targeted at Systems Administrators and power users who want to master the art of scaling and troubleshooting high-performance or highly available Graphite clusters.

If you're like me, you don't read books from front to back. If you're *really* like me, you usually don't read the Preface at all. However, on the off chance that you will see this in time, here are a few suggestions:

Online Resources

- *http://obfuscurity.com/Tags/Graphite*
- *http://graphite.readthedocs.org/en/latest/*
- *http://graphite.readthedocs.org/en/0.9.13/*
- *https://answers.launchpad.net/graphite/*

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

CHAPTER 1
What is Graphite?

This book is about monitoring with Graphite. One of my goals in teaching users how to use Graphite effectively is not *just* how to monitor "better", but how to store and retrieve data in a way that helps us manage risk, predict capacity shortfalls, and aligning our IT strategy with the overarching business needs of the organization. *Monitoring with Graphite* trains us to be able to answer more pertinent questions than "is my server alive". We should be able to quickly ascertain: "are customers suffering a degraded experience", "how has our sales team performed over the last quarter", and "do we have enough capacity to absorb a Reddit front-page posting".

Virtually every activity we perform can be measured and analyzed. Weather forecasters use climactic data to predict changes in temperature, barometric pressure, and wind speed. Sports statisticians analyze trends by players and teams to identify the most productive athletes and most effective coaching strategies. Fans of the "Adventure Time with Finn and Jake" show record the number of times that Lumpy Space Princess utters "oh my Glob" in a single season. And most important to us (I hope), software engineers and systems administrators use telemetry - recorded measurements - to measure the effectiveness of their system designs and deployments.

Sidebar What is Telemetry?

Telemetry refers to the collection of measurements, typically of remote instruments, for the purposes of monitoring. The word itself has Greek origins; *tele* means "remote" and *metron* is to "measure". For many of us, telemetry conjures up recollections of NASA space travel and the myriad of instruments spread out across the Space Shuttle cockpit or the displays in mission control centers. Although we might not be launching rockets to Mars anytime soon, it's an apt description for the sort of monitoring data we collect and act upon every day.

The type of questions we ask of our data influences the way we store, retrieve and interact with our data. Spreadsheets created with Excel, rich structured data logged to Hadoop, and *time-series* metrics saved in Graphite all have strengths and weaknesses, but were designed with a specific goal or workflow in mind. As practitioners of this information, we need to understand the use cases behind each tool in order to maximize our effectiveness while *working with our data*.

In this first chapter we're going to get a quick primer on *Time-Series* data before diving into the history and goals of the Graphite project. With this background we can better understand what differentiates it from other monitoring and trending software applications and whether it's a good fit for your organization.

If you're already familiar with the topics I've mentioned, you might want to skip ahead to Chapter Two, "Monitoring, Trending and Alerting". Although some of the topics in that chapter may be a refresher for some readers, many of the underlying principles are founded by years of personal experience building large-scale telemetry systems for companies like Heroku, GitHub and Dyn. To get the most out of this book and my presentation of the materials within, I caution you against jumping *too far* ahead.

With that caveat out of the way, let's dive in.

What is time series data?

> Mathematical!
>
> — Finn the Human

We interact with *Time Series* data virtually every day of our lives, even if we don't immediately recognize it as such. Weather reports, financial tickers and even the audio visualizer on SoundCloud use time series to convey how the things we care about, such as personal investments or the chance of rain at lunch, into an easily understood visual format that we can absorb quickly and with easily.

But what *is* time series data, *really*? Nothing more than a sequence of values collected at regular, uniformly-spaced intervals. If I asked you to step outside every hour and give me your best estimate of the current temperature in degrees Fahrenheit, the act of recording those numbers in a computer (or even a sheet of paper) would constitue Time Series data collection. Certainly this isn't the most scientific method for tracking our local weather patterns, but it's not about the accuracy of the data (in our case, anyways) but merely that we're performing the experiment regularly and uniformly.

Depending on the sort of work you're tasked with, your data may align well with traditional Time Series storage engines or perhaps even a specialized Analytics database system. From my own personal experience, *data is data*; the differences lie in the sorts of questions you wish to ask of your data.

Systems Administrators and Web Engineers are generally more interested in *rate of change* and *seasonality*, in order to quantify the health or *Quality of Service* of a particular application or host. They want to know when their services are degraded or might be trending towards a bad state.

Analysts, on the other hand, are typically looking for trends in user behavior. They are often more focused on the *distribution* of a subset of data or *metadata*. Unique events carry special meaning for them, so the ability to correlate these events using tags (or labels) allows analysts to classify users and their behaviors in such a way that pertinent questions can be asked of seemingly random data. But since you're reading this book, I'm going to assume your current job (or hobby, I won't judge you) relates more to the type of questions that are best answered with Time Series data and tools like Graphite.

Time-Series Databases

Put simply, a time-series database is a software system that's optimized for the storage and retrieval of time series data. And although nothing else I say through the rest of this book will change that, I think it's useful to talk at greater length about some of the low-hanging fruit in terms of TSDB (a popular acryonym for Time-Series Databases, and one I'll use repeatedly throughout this tome) performance and maintenance, since this is often a major factor when evaluating software trending systems like Graphite.

Storage Considerations

It's not that the average user should *need* to think about or interact directly with these data stores under normal operation, but without a database tuned and optimized to handle the workload of a high-performance data storage and retrieval system, the user experience will be dreadful. Therefore, it's important that we start with at least a nominal understanding of TSDB performance patterns. Your future self will thank me.

There's been an explosion in the number of Open Source projects and commercial monitoring products over the last few years. Advancements in Solid-State Drive (SSD) storage and the persistence of Moore's Law have made it technically and financially feasible to collect, store, retrieve, correlate and visualize huge amounts of real-time monitoring data with commodity systems (read: The Cloud). Distributed database systems (particularly NoSQL), themselves largely driven by competition in the pursuit for Big Data analytics, have made it easier than ever before to scale out horizontally and add capacity as demand necessitates.

Why do I place such an emphasis on storage? Because, without fast storage (and a lot of it), we wouldn't be able to persist all of this wonderful data to disk for later retrieval. Of course, it helps to understand what I mean by "fast storage". Frankly, it's not that hard to write to disk quickly, nor is it difficult to read from disk quickly. Where the challenge arises is doing *both at the same time*.

Prioritizing Operations

In terms of Operating System design (the software that powers your computer), the *kernel* is the brains of the operation. It's tasked with a huge variety of administrative duties, from managing available memory to prioritizing tasks to funneling data through the network and storage interfaces. Depending on the conditions involved and configuration applied, the kernel must make compromises to handle its workload as efficiently as possible. But how does this apply to us?

Remember that scene from the movie "Office Space" where the Boss walks over to his employee's cube and asks him to work over the weekend? Yeah, let's pretend we're the Boss ("Lumbergh") and the kernel is the employee ("Peter").

Lumbergh: Hello Peter, what's happening? Ummm, I'm gonna need you to go ahead and process these 20 million disk writes as soon as possible. So if you could have those done in 15 milliseconds that would be great, mmmkay?

Peter: [*sullen acceptance*]

Lumbergh: Oh, oh, and I almost forgot. Ahhh, I'm also going to need you to return a composite query for the 95th percentile of all derived metrics, immediately. Kay? Uh, and here's another 30 million disk writes. I can see you're still working on those other 20 million writes, so we need to sort of play catch-up. Kay?

Peter: [*muffled sobbing*]

Lumbergh: Thaaaanks.

This is a fictional interaction, but the demands we're placing on the kernel (and filesystems) are all too real. Writing to and reading from disk (also known as *input* and *output*, or simply *I/O*) are expensive operations if you're trying to do a *lot of one* and *any of the other* at any point in time.

A popular technique for optimizing writes is to buffer them in memory. This is fine as long as you have enough memory, but you need to flush these to persistent disk at routine intervals or risk losing your data in the event of a system failure.

On the other hand, it's very effective to use in-memory caches to keep "hot copies" of data for queries (reads). Again, this is an effective approach as long as you have enough memory and you expire your cached answers frequently enough to ensure accurate results for your users.

Graphite uses both of these techniques to aid the performance of its underlying time-series database components:

Carbon
: A network service that listens for inbound metric submissions. It stores the metrics temporarily in a memory buffer-cache for a brief period before flushing them to disk in the form of the Whisper database format.

Whisper
: The specification of the database file layout, including metadata and *rollup* archives, as well as the programming *library* that both Carbon and the Graphite web application use to interact with the respective database files.

We'll touch on both of these components in greater depth later on. For now it's enough to understand their respective roles in the Graphite architecture.

The ability to store **and** retrieve time-series data quickly, and at high volume, is key to the success of Graphite. Without the ability to scale, it would be impractical to use Graphite for anything outside of small teams and hobby projects. Thanks to the design of Carbon and Whisper, we can build significant clusters capable of processing many millions of datapoints per second, making it a suitable visualization tool for virtually any scenario where time-series analysis is needed.

What is the history of Graphite?

Once upon a time, years even before the invention of Nagios, a Swiss man named Tobias Oetiker worked at the De Montfort University in Leicester, UK. Looking for a method to track network activity on their lone Internet connection, Tobias developed a small Perl script for monitoring traffic levels on their network router. Querying SNMP interface statistics every five minutes, it would use this data to generate a series of graphs detailing current and past network levels.

This tool came to be known as the Multi Router Traffic Grapher (MRTG). If you worked for an Internet Service Provider or telcommunications company in the 1990's or 2000's, there's a good chance you were exposed to this tool or it's spinoff, the Round-Robin Database (RRD). I was fortunate to work for early Internet companies like Digex and Cidera where it became ubiquitous, and it almost certainly set the foundation for my early interest in monitoring practices and technologies.

Years later, Chris Davis, an engineer at Orbitz, the online travel agency, began piecemeal development on the components that would later become known as Graphite. The rendering engine began like so many other great hacks; he just wanted to see if he *could build one*. It was designed to read RRD files and render graphs using a URL-based API.

Whisper, the database library, was a desperate attempt to fix an urgent bug with RRD before a critical launch date. At the time, RRD was unable to accept out-of-sequence data; if metric *A* with a timestamp of 09:05:00 arrived after metric *B* with a timestamp of 09:05:30, the former metric would be completely discarded by RRD (this was later redesigned). Whisper addressed this design shortcoming specifically, and at the same time, drastically simplified the configuration and layout of retention periods within each database file.

The Carbon service marked the introduction of a simple network interface abstraction on top of Whisper that enabled anyone with a computer to submit metrics easily. It evolved to add an in-memory buffer cache and query handler, addressing both performance and the need for real-time query results; `carbon-relay`, a daemon capable of load-balancing or replicating metrics across a pool of *carbon-cache* processes; and `carbon-aggregator`, built to aggregate individual metrics into new, composite metrics.

By 2008, Chris was allowed to release Graphite as Open Source. Before long it was mentioned in a CNET article, then on Slashdot, at which point adoption took off. In the years since, an entire cottage industry has built up around the Graphite API and countless businesses rely on it as their primary graphing system for operational, engineering, and business metrics.

What makes Graphite unique?

While Graphite continues to evolve and add new features routinely, much of its success stems from its adherence to simple interfaces and formats, resulting in a shallow learning curve for new users. Metrics are easy to push to Graphite, either from scripts, applications or a command-line terminal. Graphs are crafted from the URL-based API, allowing them to be easily embedded in websites or dashboards. The web interface empowers users to prototype graphs quickly and immediately, with virtually no training or formal instruction necessary. And a huge community of users among a range of diverse backgrounds and industries means that new and unique rendering functions are always being contributed back upstream to the project.

Simple Metrics Format

Arguably one of Graphite's most important attributes, the metrics format for submitting data to the Carbon listener is beautifully simple. All that's required is a dot-delimited metric name (e.g. "foo.bar"), a numeric value, and an *epoch timestamp*. Instrumenting your application to send telemetry can be done in a couple short of lines of code, or if you're impatient, with the help of some basic UNIX commands in your shell terminal.

Example 1-1. Sending a metric value from the command-line

```
$ echo "foo.bar 41 `date +%s`" | nc graphite-server.example.com 2003
```

In this example we're using the `echo` command to print the metric string "foo.bar" along with my favorite prime number, and an epoch timestamp generated by the `date` command. The output is piped into netcat (`nc`), a little network utility that connects to our imaginary Carbon service at "graphite-server.example.com" TCP port 2003, and sends our data string. The trailing newline character provided by `echo` notifies the Carbon server that we're finished and the connection can be closed.

Epoch Timestamps

An epoch timestamp represents the number of seconds in time that have passed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not including leap seconds. This might be useful the next time your UNIX-friendly significant other asks you for the epoch time on Valentine's Day 2014 (1392354000).

Graphing API

Unlike its predecessors and many of its contemporaries, Graphite doesn't rely on static configurations or batch jobs to create new graphs. All of its data rendering, from its traditional PNG charts found in the web interface, to the JSON output used by client-side libraries to compose elaborate dashboards and information graphics, is constructed on-the-fly using its comprehensive API.

Every single graphing feature available to Graphite users is exposed via this API, precisely *because* its web UI consumes the very same interface. For Graphite developers and users, this is the ultimate case of "eating one's own dogfood".

Of course, this makes sharing graphs with anyone as easy as sharing the URL that constructed it. Even better, that same URL can be embedded in your company dashboards or even a website. As long as the person loading the graph has network access (and in the case of password protection, the necessary credentials) to the webserver running your Graphite application, they should be able to see *exactly the same data* that you're viewing.

Thanks to the popularity and stability of Graphite's API, a huge ecosystem of third-party tools and services have grown up around it. In fact, not only do a wide variety of applications consume the Graphite API, but some projects have even developed *backend service adapters* to enable Graphite to speak to their own storage systems.

Rapid Prototyping

As fantastic as the Graphite API is (and it *really is*), most users' first encounter with Graphite is the web interface, and more specifically, the Composer. While nobody's going to confuse its design for something released by Apple and Jony Ive, it does a great job where it matters: navigating metrics and *saved graphs*, adding to and removing metrics from graphs, applying statistical transformations, and exposing the full breadth of the rendering API.

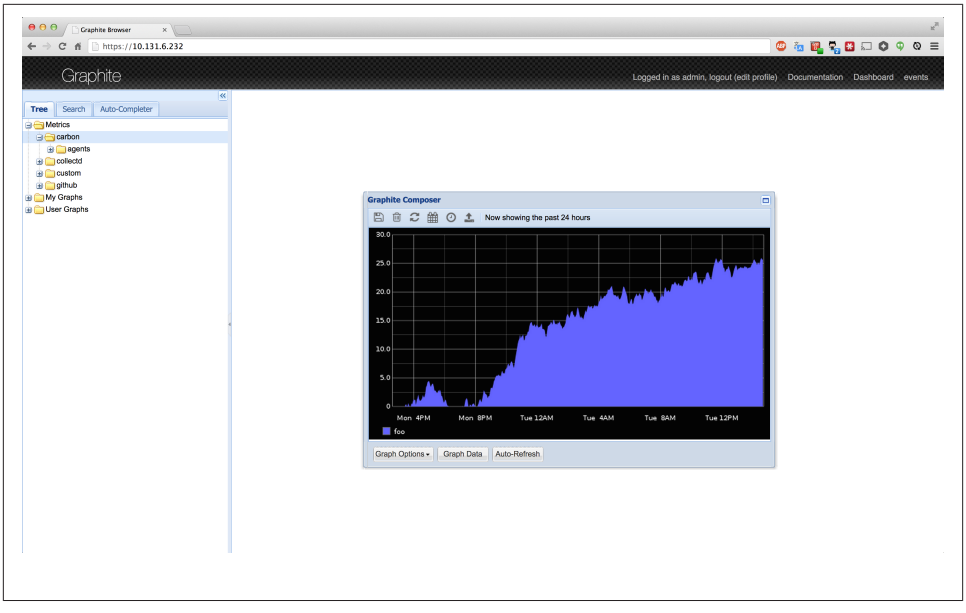


Figure 1-1. Composer window

When talking about everything the Composer is capable of, it's easy to get lost in the woods and lose track of our mission: visualizing and correlating our data. Fortunately, the utilitarian nature of the Composer's interface makes it perfectly suited to doing just that. Navigating the *Metrics Tree* of nested folders and metric names is a familiar experience. Clicking on a metric name adds its data series to the *Graphite Composer* window frame in the center of the screen. Transformative functions are easily applied from the *Graph Data* dialog window, resulting in new and exciting ways of interpreting the data. Switching from a *Line Chart* to an *Area Chart* is one click away in the *Graph Options* menu.

Best of all, the graph automatically refreshes after every action to reveal the new arrangement. Feedback is instantaneous and intuitive. Graph URLs are easily copied and shared with your peers. Those same people can make adjustments and pass back the new URLs, which are just as easily loaded back into your Composer again. Before long you've mastered a workflow that empowers you to rapidly isolate anomalous behavior and identify causal relationships in a manner that's lacking in most traditional monitoring and trending systems.

Rich Statistical Library

But perhaps more than anything else, what really distinguishes Graphite from every other commercial or open source monitoring system out there is its exhaustive library of statistical and transformative rendering functions. As of version 0.9.13 there are 88

documented functions used to transform, combine or perform statistical computations on series data using the render API.

You probably won't be surprised to see that Graphite includes primitives to help identify the min, max or mean of a series. You might even be pleased to see that it can calculate the *percentiles* and *standard deviation* of your series. But I think you'll be truly impressed with its breadth of coverage when you discover its support for various Holt-Winters forecasting algorithms and virtually every sorting and filtering criteria you could imagine.

How is this possible? The Graphite community is wonderfully large and varied. Users represent a number of different industries and vocations: Software Engineers track application performance and profiling data; Systems Administrators feed in server data to track resource utilization; Marketers monitor user activity and campaign results; Business Executives correlate quarterly results with Sales numbers and Operational expenses. At one time or another, every one of Graphite's rendering functions was contributed by someone who discovered a new use case or algorithm for their task, developed a function to process the data as required, and then submitted this enhancement back to the Graphite project in typical OSS fashion.

Chained Functions

Graphite treats each series on a chart as a stream of data: the original raw data can be passed into a rendering function; the output of that function can then be passed onto another function; Lather, Rinse, Repeat! Once it has determined there are no additional transformations, Graphite passes the series on to your choice of rendering formats (e.g. graph, CSV or JSON response, etc).

Some time ago, I worked on the Operations team at Heroku, managing a huge fleet of Amazon Elastic Compute Cloud (EC2) instances in production. As time went on we began to notice an abnormally high rate of system failures for a particular type of instance. Fortunately we were able to instrument our platform services to fire off a Graphite metric (`ec2.instances._hostname_.killed`) every time one of these instances became unresponsive, forcing us to terminate and replace it in the cluster. The metric was pre-configured with a *resolution* of 15 minutes for up to one year.

What is Resolution?

When describing time-series data, the *resolution* of a metric describes the precision and interval for which the metric's values are captured. For example, Carbon reports its own internal statistics at a resolution of one minute for up to one year. The precision is one minute, meaning that each datapoint in its Whisper database represents a 60-second interval. It will store enough of these datapoints (525,600, to be precise) to describe one year's worth of data (not counting leap years).

We can always request *lower precision* values using `summarize()` but we can never ask for *higher precision* values than our database was configured to support at that resolution.

By chaining together some common render functions, we were able to construct a *target* query string that counted the number of failed instances and summarized the results into 24-hour buckets. Most render functions accept one or more series (or a wildcard series) and one or more arguments within parentheses. Each subsequent function wraps around the one preceding it, resulting in a nested sequence of input and output. The end result would've looked something very much like the query in ???.

```
target=summarize(sumSeries(ec2.instances.*.killed), "1d")
```

In this example we used the `sumSeries()` function to aggregate all of the matching metric series described by our wildcard metric string, `ec2.instances.*.killed`. This returned a single series representing the sum of all terminated instances. However, our Amazon representative asked us to report our counts as a daily total, so we took that result and fed it into the `summarize()` function (with the `"1d"` interval argument) to group our results into 24-hour resolution windows.

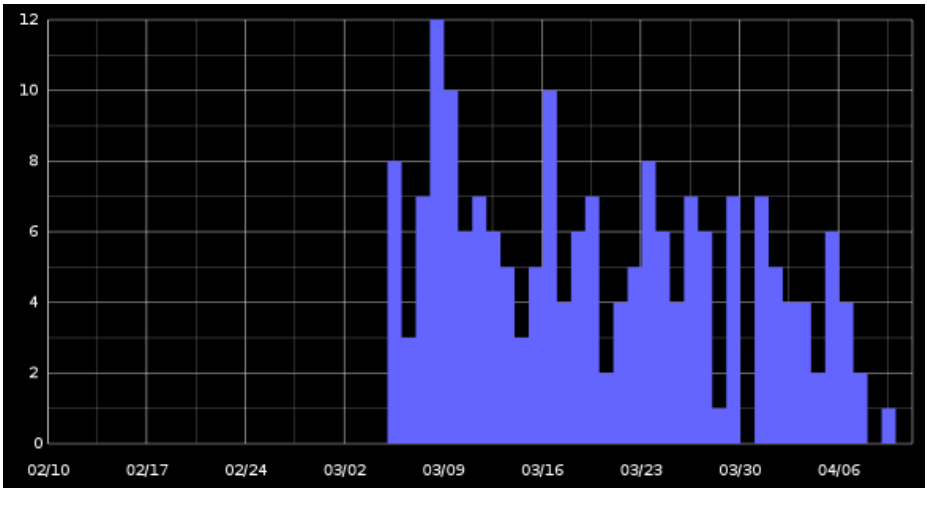


Figure 1-2. Calculating the number of failed EC2 instances reported daily

What is Seasonality?

Seasonality is often used to describe the characteristics of regular or predictable patterns which recur around the same time every year. If you live around the Washington DC area as I do, you know to expect muggy weather in August, snow in January, and the bloom of cherry blossom trees in April. This is seasonality, and it's often used to describe the cycles in retail sales (e.g. Black Friday), weather patterns, and even website visits and server performance.

Many people use the terms *seasonality* and *periodicity* interchangeably, and you're generally safe to do so as well. Personally, I prefer the term periodicity when talking about time-series data since we're frequently comparing periods smaller than calendar years or even seasons. If you're an online retailer or publisher, it's very common to track cyclical patterns in your data on a weekly, daily or even hourly basis.

Case Studies: Who uses Graphite in production?

There's a seemingly infinite number of tasks that Graphite is well suited to handle. Because Graphite makes it so easy to store and retrieve virtually any type of numeric data, you're almost as likely to find Chief Executives or Business Development Analysts using it as you would Systems Administrators and Web Developers. But rather than take my word for it, let's hear from a few businesses that are using Graphite to power everything from application and security monitoring to business decisions.

Booking.com

Like Orbitz, the company behind Graphite, Booking.com is one of the Internet's busiest online travel agencies. They also happen to run one of the world's largest Graphite installations. According to Brad Lhotsky, Systems Security Team Lead, all of Booking.com's primary system, network, and application performance data is stored in Graphite, alongside segments of their security and business intelligence.

"We track both technical and business metrics in Graphite. It allows us to understand the business costs of outages or performance degradations easily. It also allows us to correlate business trends with system, network, and applicaiton level trends. This leads to more informed decision making by the technical people and confidence in the technical teams by the business folks."

"The scope of the business level monitoring we do with Graphite is significant. Most technical decisions are made only once business value is understood. Graphite provides a convenient way for technical and non-technical people to access both the technical and business data in an easy to understand way."

Graphite provides a simple API that enables their Application Developers, Systems Administrators, Network Engineers, and Security Engineers to store time-series data at will. This means they can correlate data across all these disciplines to understand the full impact of code changes, network infrastructure changes, or security policies in near real-time.

GitHub

Millions of people across the world use GitHub to help them manage their software development pipeline and collaborate with like-minded users. Graphite provides GitHub's engineering and operations teams with the tools they require to collect and correlate massive amounts of time-series data. Scott Sanders, Infastructure Engineer at GitHub, describes how they use Graphite to comprehensively monitor their host, application, service, business, and financial metrics.

"In our operations team we use Graphite to drive many of our alerting systems and perform capacity analysis. It's also heavily used by our development teams to analyze the performance impact of changes as they're implemented and slowly deployed across our infrastructure. Our business and sales teams use Graphite to track impact and effectiveness of their campaigns and initiatives."

"We interact with a large portion of our graphs through [ChatOps]. A culture of data driven investigation through chat has enabled us to interact with our data in a manner that is visible to the entire team participating. In addition this creates an audit log allowing us to assess our successes and failures well after the event has ended."

Graphite's scalability and flexibility allows it to be used as a common utility across all departments within GitHub, making it that much easier for teams to collaborate and share information using the same key concepts and interfaces.

Etsy

Etsy is an online marketplace bringing together artists, craftspeople, and consumers together to buy and sell unique and homemade goods. Daniel Schauenberg, Senior Software Engineer at Etsy, describes how Graphite fits into their self-service engineering culture to measure and track website performance and customer experience.

"We use Graphite heavily to track and monitor etsy.com application metrics. This includes performance of different things like page render times or database requests. These timings come from StatsD which is our biggest ingress point for Graphite data. We also use counts in there heavily to get the number of e.g. logins or checkouts at any given point."

"Graphite makes it really easy to create metrics ad hoc. This means a developer can add instrumentation to their feature with a single line of code and immediately get feedback.

It is very self service and doesn't require anyone to enable something or explicitly grant access."

Electronic Arts

One of the largest and most successful video game publishers in the world, Electronic Arts (EA) has to track performance data for hundreds of millions of customers. Steve Keller, Systems Architect, Monitoring at Electronic Arts, talks about how EA uses Graphite to gain insight into platform and user telemetry at a significant scale.

"I first started using Graphite about 3.5 years ago to graph data from our monitoring systems. Within a year, I found other teams at EA had begun to use Graphite for collection of internal metrics. We eventually created a larger Graphite infrastructure to combine data from all sources, and now my team manages several Graphite clusters for multiple teams at EA."

"Today, countless dashboards and real-time monitoring consoles have been created for EA around the world; we rely on Graphite to give us a very important view into systems performance and the player experience."

Why should I use Graphite?

Whether your job title is Systems Administrator, Software Developer, QA Engineer, or CEO, it's crucial that your systems and applications are measured accurately and continuously. Without real-time monitoring and a high-performance analytical datastore, we lack the perspective to qualify our current, past, or future performance. Business decisions are increasingly *data-driven*, and Graphite provides all the tools to help users *collect*, *store*, *retrieve* and *analyze* data quickly and effectively.

If this sounds like something that piques your interest, I urge you to grab a caffeinated drink and read on. It's about to get real up in here.

CHAPTER 2

Monitoring Conventions

Everyone seems to have a different definition for what *Monitoring* means to them. Many folks know it as a convential *polling* system like Nagios. For others it might mean walking their networks with SNMP and Cacti, or perhaps even a bespoke collection of Perl scripts and artisinal cron jobs. Some companies don't run internal monitoring systems at all, perferring to outsource all or part of their monitoring to hosted monitoring services. No matter which tools you cobble together or how you orchestrate them, most of these systems cover a common set of operational responsibilities.

From my experiences, it's important that we, the maintainers and users of these *Monitoring Architectures*, share a common vocabulary and understanding of the logical areas of functionality that make up these systems. Describing to your peers how you "*instrument* your application *telemetry* and *aggregate* the results (because they report irregularly) before firing them off to your *trending* system for *correlation* and *fault detection*" is almost certainly going to explain more about your setup than "we monitor stuff".

Don't get me wrong, I'm all for brevity, but words really do matter. And for better or worse, we've got enough of them to choke a horse. Although those two expressions may in theory be saying roughly the same thing, the former tells us a lot more about what you do and how it adds value to your organization. Above all else, being able to speak lucidly about your monitoring systems can go a long way towards building trust with your customers, your engineering and operations teams, and your business leadership.

Three Tenets of Monitoring

Monitoring is a generic way to describe the collection of software and processes we use to ensure the availability and health of our IT systems and services. In an abstract sense, monitoring can be broken up into three main categories: *Fault Detection*, *Alerting*, and *Capacity Planning*. Each of these can be dissected even further into specific functional tasks, but we'll get to that later. For now let's take a brief look at these broader concepts

since they form the basis for most of the legacy approaches among monitoring vendors and Open Source projects.

Fault Detection

The primary goal of any monitoring system should be to identify when a resource (or collection of resources) becomes unavailable or starts to perform poorly. We use *Fault Detection* techniques to compare the current state of an asset against a known good (or simply *operational*) state. Traditionally we employ *thresholds* to recognize the delta in a system's behavior.

For example, we might want to know when our webserver's "time to first byte" (i.e. the amount of time it takes for a client to download the first byte of content from a web site or application) exceeds 300 milliseconds. Our monitoring software should be able to review the metric - either by polling the service directly or reviewing the data provided from a dedicated collection agent, detect when the webserver becomes sluggish (typically by polling the service directly), classify it as a fault, and escalate the *Alert* to the responsible parties (usually a very tired on-call person at 3am).

With the emergence of Big Data and other data science pursuits, we're starting to see new and exciting *anomaly detection* techniques and algorithms applied to the area of IT monitoring. Basic thresholds are very much a practice in *trial and error*. Conversely, these new approaches attempt to leverage the advances in Machine Learning to automate fault detection software to be more intelligent and proactive, and ideally, result in fewer false alarms.

False Alarms

You're bound to hear monitoring folks refer to false positives or negatives when talking about fault detection and alerting behavior. A *false positive* is an alert triggered by our monitoring threshold, either by accident or misconfiguration. Not only are these annoying, but they can quickly lead to *pager fatigue* and a general distrust of the monitoring system.

On the other hand, *false negatives* are alerts that our monitoring system fails to detect. These are usually caused by improper thresholds, but can also be triggered by improper check intervals (running at the wrong times or too infrequently), or simply a lack of proper checks. Unfortunately, false negatives are usually identified too late, as a consequence of a host or service outage.

Alerting

> If my answers frighten you then you should cease asking scary questions.
>
> — Jules Winnfield

If you've been in IT for a while, there's a good chance you've had to carry a pager after business hours. Even worse, it's probably woken you (and your significant other) in the middle of a lovely dream, or interrupted a fun night out with your friends. I don't know anyone that enjoys *Alerting*, but I think we can all acknowledge that it's a necessary part of monitoring.

For all practial purposes, alerting constitutes the moment that your monitoring system identifies a fault that demands some further action. In most cases the system is designed to send an email or page to a member of your engineering or operations teams. But getting your alert to its destination is not always as simple as it sounds. Sometimes the recipient may be out of range of cellular service or simply have their ringer turned off (accidentally, I'm sure). This is why we should also include on-call scheduling, notification routing, and escalations in any discussion of alerting.

When your team grows large enough to "pass the pager around", it's a good time to start looking at various on-call scheduling approaches. It's common to see team members trade on-call responsibilities every week, every few days, or even *every day*, depending on the severity and "pain levels" of your rotation. Ideally, you want the handoff to occur during normal business hours when both individuals are awake, alert (excuse the pun), and the person coming off their rotation can pass along valuable information from their shift.

The good news is that as the size of your team increases, the amount of time each person has to cover on-call decreases proportionately. The bad news is that on-call scheduling software has traditionally been pretty awful. Until the last few years, most companies had to resort to manually updating their alerting configuration with the new recipient and any escalation paths. If you were lucky, someone wrote a script to manage this automatically, but these were susceptible to bugs and typos (like all software), often resulting in false negatives.

These days, it's much more common to see companies route their alerts through an external alert service like PagerDuty. A cottage industry of alert management companies have emerged over the last few years, offering notification routing, escalation, and on-call scheduling services. As someone whose lived through the lean years before outsourced alert management, I strongly recommend investing in any of these services or even their Open Source counterparts.

Capacity Planning

Anyone whose ever been asked to predict the growth of their application, customer base, or the budget for next year's IT purchases has been faced with the need for *Capacity Planning*. Heck, even the act of being alerted when your server's disk hits 90% capacity is a simple version of capacity planning (albeit a very poor one). This might not be something you're asked to do very often (or at all), but if you're tracking your server

and application metrics in Graphite, there's a good chance you'll be prepared when the opportunity presents itself.

The act of capacity planning is really just the ability to studying trends in your data and use that knowledge to make informed decisions for adding capacity now, or in the near future. Time-series data plays a powerful role in executing your capacity planning strategy successfully. If you intend to use Graphite for this (and why wouldn't you), at some point you'll face the tradeoff between storing your metrics for as long as possible and the increased disk space requirements needed for supporting a lengthy retention policy.

Obviously, nobody wants to keep their data around for a few years because they think it's going to be useful for current troubleshooting efforts; they use these older metrics to observe annual trends in our systems and business. Although this typically means *downsampling* your metrics to save on disk space, having this data available (even at lower precision) is invaluable for studying long-term trends. In fact, it's not uncommon to see users with ten-year retention policies!

We're not going to cover capacity planning at depth in this book, but it's important to acknowledge its purpose and understand its relationship with the sort of time-series data you'll collect with Graphite. For a proper treatment on the topic, I highly recommend John Allspaw's *The Art of Capacity Planning*, and *Guerrilla Capacity Planning* by Dr. Neil J. Gunther.

In most circumstances you'll hear me refer to *Trending* since we're more interested in the broad application of storing, retrieving and analyzing time-series data, but rest assured Capacity Planning is always hiding in the shadows, ready to pounce at a moment's notice.

Rethinking the Poll/Pull Model

Having a shared vocabulary of logical monitoring systems is important, but one of the biggest advancements in recent history has been the deconstruction of the monolithic monitoring system into discrete functional components. As a result, we understand our functional competencies and responsibilities better than ever before. But perhaps one of the biggest developments is the explosion of new tools and frameworks in the monitoring space.

For years, businesses were resigned to the fact that Nagios was effectively the only choice in Open Source tools for host and service monitoring. Because it was "good enough" for most use cases, there was never enough "operational angst" to drive innovation in the space.

If we look back over the last five years we can see parallels between the development of *Service Oriented Architectures* (more recently re-coined as *Microservices*) and the advancement of Open Source monitoring tools. Users and developers began to recognize

that you didn't have to reinvent the entire wheel; it was enough to build and publish small, sharp tools that emulated the legacy interfaces and covered specific functional areas: alerting, notifications, graphs, etc. These days you'll be hard pressed to find many businesses that don't use a variety components to form their monitoring architecture.

While it's true that most CIOs would prefer the simplicity of a single vendor that offers the traditional monolithic "Enterprise Monitoring Suite", the reality is that all businesses are different and there is no such thing as "one size fits all" when it comes to monitoring systems. If you take away nothing else from this book, please remember that axiom. I promise it will save you (and those you care about) plenty of tears and heartache.

Before we dive into the features of a modern monitoring stack, I want you to forget everything you think you know about monitoring systems. Whether you're the CTO of a San Francisco-based startup or a battle-tested Systems Administrator working on government contracts, you probably have a lot of preconceptions about what a monitoring system *should* look like. Try to forget, at least for a bit, that you ever heard of Nagios and NRPE; I'm about to free your mind.

Pull Model

> Then you'll see, that it is not the spoon that bends, it is only yourself.
>
> — Spoon Boy

The traditional approach to IT monitoring centers around a *polling* agent that spends a great deal of time and resources connecting to remote servers or appliances in an effort to determine their current status: are they reachable on the network with *ping*; what does their SNMP output tell me about their *CPU usage*; can I execute a remote command on that host and interpret the results?

This is what we generally refer to as a *pull model* in that we're pulling (or polling) the target systems at regular intervals. Historically, we've asked very simple questions of our systems ("is it alive") and organized our operations staff and monitoring software around the goal of minimizing downtime. Sadly, this characterization reinforced the legacy perception of the IT department as a cost center.

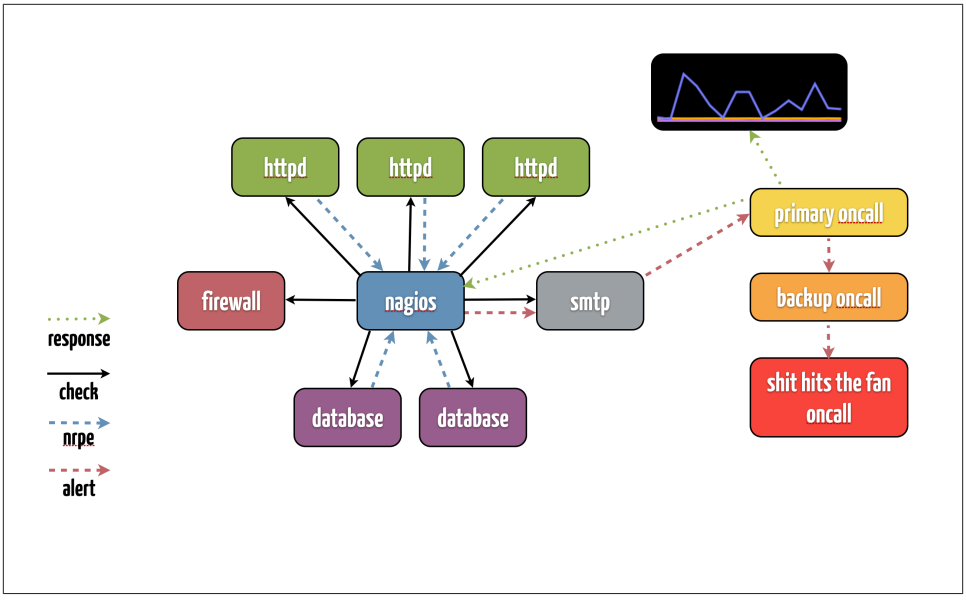


Figure 2-1. Legacy Pull Model

A number of factors have caused IT departments and Operations teams to start being viewed in a much more positive (and lucrative) light. Thanks to the popularity of elastic computing resources ("The Cloud"), improved automation through configuration management, and our increased reliance on Software as a Service (SaaS), businesses have begun leveraging their IT investment into cost efficiencies in other departments. In many ways, the modern Operations team can be as much of a profit center as the traditional sales channel (ok, maybe not *quite as much*, but you get the idea).

Of course, this means that we need to measure our IT performance with vigor and real qualitative numbers. It's not that we aren't already doing this, but in many cases we treat *Monitoring* and *Trending* as two completely distinct stacks of information. Businesses deploy "Enterprise Software X" to monitor uptime and general system health, and then deploy a separate instance of "Enterprise Software Y" to gather SNMP data and render graphs. There's still a massive amount of duplication of effort; not to mention all the Nagios performance data that's frequently discarded (the default behavior for Nagios).

Push Model

Fortunately, many companies are beginning to recognize the folly in this model. The performance data previously lost is now collected and stored at high precision and used to power questions relating to availability and quality of service. Metrics are *pushed* from their sources to a unified storage repository, arming us with a consolidated set of

data that we can use to drive both our IT responses *and* our business decisions. We can use the exact same metrics to measure the health and availability of our systems.

There's a great deal of flexibility that comes with instrumenting our systems to *send* telemetry data, rather than pulling it manually. Collection tasks are decentralized, so we no longer need to scale our collection systems vertically as our architecture scales horizontally (i.e. to many nodes). Systems report in as they're available; no need to deal with timeouts, disconnects or retry attempts. Each system can use the transport mechanism best suited to their design or environment; servers can be tooled to leverage TCP sockets, message queues, or plain old log streams.

But one of my favorite aspects of this *push model* is that we can begin isolating the functional responsibilities of our monitoring systems. We're no longer forced to deal with a monolithic "black box" to manage our IT assets. In fact, as we begin to understand these discrete functional units, we can begin looking to other industries (e.g. airlines, medicine, etc) and apply their best practices to our own.

Where does Graphite Fit into the Picture?

Glad you asked. Truth be told, Graphite often comes into play at almost every step of the monitoring lifecycle. Does this mean Graphite is one of those terrible monolithic applications I mentioned earlier? Not at all. But if you're in a position to use Graphite as your "source of truth", it is capable of fulfilling different roles at various points of the monitoring architecture, thanks to it's flexible design, well-defined service interfaces and API coverage.

Having a centralized, canonical system of record for all metrics and state is a key part of any responsible and trustworthy monitoring architecture. In legacy systems it was common to store network accounting data in one system, host and service monitoring state in another system, and capacity planning telemetry in yet another. This lack of a unified source of truth often resulted in conflicting information, or worse, the inability (or great difficulty) to correlate disparate data sources.

Now there are inevitably times when it makes sense to have some overlap (or even duplication) of measurement data, particularly when the types of questions asked of the data vary significantly. Be careful not to fall into the trap where you feel you must choose one tool for *all of your data*. For example, it often makes sense to keep rich analytical data in a Hadoop cluster or business-related data in an Excel spreadsheet. From my experience, the questions you need to ask of your data should influence the tools you use to store them, and not the other way around.

Regardless, there will eventually come a time when you need to correlate your IT and business metrics. Having a unified source of truth in a service like Graphite will make quick work of those questions that would often be difficult and time-intensive to answer otherwise. Powering your alert responses and decision-making with time-series data

gives you a level of intelligence and perspective that simply isn't possible with "ping" checks.

But please keep in mind that the discussion of pull-vs-push models is completely orthogonal to the need for proper time-series collection and storage. No matter what your monitoring systems resemble, you should *absolutely* have a data visualization service like Graphite at your disposal, for all the reasons I've laid out previously. I feel strongly about designing your monitoring systems properly, and that's why I've dedicated almost half of this chapter to a discussion about the topic. Tough love.

In the next section we'll describe a modern monitoring architecture based on the push model. I'm going to explain each functional area and try to give practical examples for each. But more importantly, we'll discuss how Graphite relates to each in order to give you a better understanding of its versatility and how vital a proper time-series engine is to each layer of the monitoring stack.

Composable Monitoring Systems

If you've been in the IT industry for long, there's a very good chance that you've had to procure or evaluate a commercial vendor's software product. Their glossy brochure or website almost certainly checked off all the requirements on your specification and promised a one-stop comphrehensive "solution". The overly-attached sales rep hounded you incessantly, and you finally relented.

Inevitably, your productivity took a hit because one or more features wasn't quite the right fit, didn't work as described, or worse yet, didn't exist at all (despite claims to the contrary). If you're lucky, you got a direct response from your rep (who's no longer nearly as needy, for some reason) who reached out to their engineering team and got a roadmap estimate for sometime in the next 6-12 months. If you *weren't* lucky, you got stuck in a confusing loop of Knowledge Base articles and community forums.

Weeks turned into months, months grinding into years, and before long you realize that you're fortunate because someone else took over your old role while you transitioned into a nice dark cubicle with a red stapler and a job managing the nightly tape backups.

Now, consider the alternative - an ecosystem of utilities and services that work as building blocks, allowing you to assemble a monitoring architecture that's custom-tailored to the specific needs of your organization. Components can more easily be upgraded, replaced, or enhanced without the lock-in usually associated with single-source vendors or monolithic software products. In many cases, you can run competing "blocks" in parallel for evaluation with the help of load-balancing software and related bridge services.

In short, a Composable Monitoring System offers you a level of *flexibility* that simply can't be matched by any other approach.

Even if you have no desire to deploy and manage a multi-faceted composable monitoring system, it helps to understand the functional areas involved. Most existing monitoring software products are built on these discrete components, even if the user interface completely abstracts the underlying gook from the user.

The components I'm about to introduce are based on well-established patterns common to both Open Source projects and commercial services. Having a solid grasp of these concepts will help you to better understand how all of these different products work and makes you a more educated "consumer".

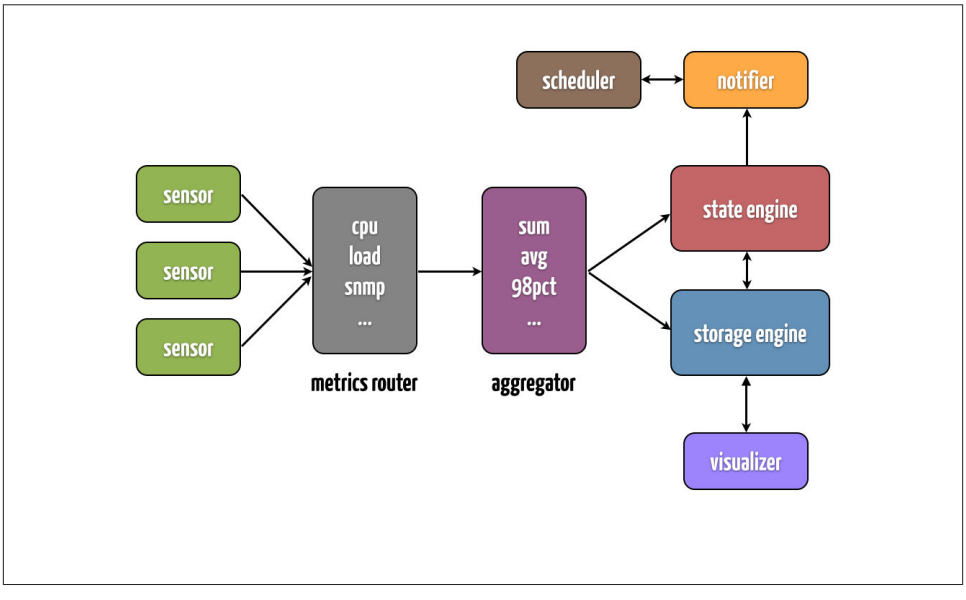


Figure 2-2. Composable Monitoring System

The diagram in Figure 2-1 demonstrates a typical workflow for data and interactions within a modular monitoring system. Metrics generally flow in a unidirectional pattern, from collection of sensor or emitter data (i.e. *telemetry*) to *aggregation*, into a *state engine* for tracking state and thresholds, and eventually to a *storage engine* for long-term archival and retrieval. Along the way we may fire off *notifications*, and users are almost certainly going to use *visualization* for troubleshooting and correlation activities.

Your environment may dictate a different topology or information flow, and that's fine. Your monitoring system is much more likely to be influenced by your existing IT systems design (as it should be) than my incoherent ramblings. Although the monitoring principles I espouse here are heavily influenced by real-life production architectures at scale, it's entirely possible that your environment and circumstances are sufficiently different as to render my suggestions moot. In this case I urge you to listen to your own experi-

ences but study the functional separation of these systems, because the logical components I've described here are *universal*.

Telemetry

> Look, do you wanna play blind man? Go walk with the shepherd. But me, my eyes are wide fucking open.
>
> — Jules Winnfield

Collection Agents and Sensors

For operating systems, network services, and even hardware devices, it's still common to see many companies using SNMP to collect and expose (or even extend, using custom shell scripts) their host-level telemetry. Because they're able to be emitted or polled at regular intervals, it's possible to send them directly to the storage system, bypassing any sort of aggregation or normalization. Unfortunately, SNMP has a negative connotation for many users due to it's archaic configuration, confusing usage, and questionable security record.

SNMP is not going anywhere soon, at least with regards to network devices. Manufacturers are starting to offer the ability to export metrics, typically using packet sampling protocols like NetFlow or sFlow. Projects like Evenflow (*https://github.com/obfuscurity/evenflow*) provide a bridge service for accepting data from these types of exports into your metrics stream.

Personally, I'm a big fan of the collectd project (*http://collectd.org/*). It's a highly extensible collection agent that works well on all Linux distributions and BSD systems. Installation and configuration is easy and the default setup provides a great out-of-the-box experience. Written in C, it performs significantly better and more reliably than competing collectors. Best of all, it has a huge collection of supported input and output plugins, including support for sending metrics directly to Graphite's Carbon listener.

Diamond (*https://github.com/BrightcoveOS/Diamond*) is another popular collection agent among the Graphite user community. Because it's written in Python, many Graphite (another Python app) developers and users find it a better fit for writing new plugins from scratch. Regardless of the host-level collection agent you select, it's important that you use one that has a solid breadth of coverage for your particular environment.

Application Instrumentation

Developers, QA engineers and even Operations teams rely on software instrumentation to report telemetry data for tracking performance and identifying bugs in the code they

release to production. Without these measurements, there's no simple method for accurately tracing a problem back to its source, or correlating causal relationships between systems or services. Proper instrumentation can quickly isolate regressions between software versions or deployments.

StatsD (*https://github.com/etsy/statsd/*) is a very popular aggregation daemon created by engineers at Etsy to aggregate and normalize metrics before forwarding them to a time-series backend like Graphite. We'll touch more about it's service-side benefits in the Aggregation section, but it's an important technology to acknowledge in terms of application instrumentation. StatsD libraries are plentiful, offering language bindings for virtually every modern high-level programming language in use on the Web today.

Using a StatsD client library, developers can fire off new measurements with a single line of code. It supports *timers*, *counters*, and *gauges* natively, making it trivial to track durations, cumulative sums, or rates over time. And perhaps best of all, StatsD is designed to use the non-blocking and stateless UDP protocol by default. This means that clients will never have to wait on a StatsD request to complete; it doesn't *block* progress of your code because of e.g. a remote connection timeout or processing request. This is a significant consideration for developers wanting to collect performance data on their applications at "Web Scale".

```
StatsD::increment("widgets.sold");
```

Logging

It's worth mentioning that logging is a completely valid alternative to dedicated instrumentation libraries. Many companies already have a robust logging pipeline available; this diminishes the need for a dedicated aggregation service like StatsD or retooling your applications with the libraries needed to speak to it.

In fact, the entire Heroku platform emits telemetry data using a standard log format, leveraging their high-performance distributed log routing service. Although the logs inevitably get stored in a high-volume log archival service further down the line, they "drain" metrics from this "log stream", applying aggregation rules and forwarding it on to their time-series storage system.

If you can print your metrics to `STDOUT` in a predictable and structured format, there are a variety of logging tools that will happily parse your entries and convert them into Graphite-compatible metrics. As we'll see later in the *Event Stream* and *Aggregation* sections, a number of companies use this approach to leverage their existing logging infrastructure.

```
measure#cache.get=4ms measure.db.get=50ms measure#cache.put=4ms
```

Business Telemetry

I frequently get asked by colleagues which metrics they should track in Graphite or related systems. Among the usual suspects of host-level and application-level metrics, I strongly encourage everyone to track their Key Performance Indictators (KPI). Unfortunately, there really is no universal KPI that I can blindly recommend; they're going to be unique to your business or organization.

If you're an online retailer, it's a no-brainer to track metrics related to sales and shipments. These reflect the current health of an online business, but they're not a useful predictor of future trends. Instead, think about the relationships and experiences that may cause your business to falter. Are your customers happy? How can you tell? If you're using an online ticketing service, they almost certainly offer an API where you can track your customer support levels and convert them into time-series data for Graphite. This sort of KPI may one day help you explain a downturn in customer retention levels or even new signups.

Business telemetry is not only useful for predicting trends. Often, we want to be able to measure the effects of a service outage in real dollars. Without metrics that describe the health of your sales pipeline or revenue numbers, there's no *quick* method for correlating these events. The engineer that manages to capture this sort of data and present it on an executive dashboard is going to be viewed in a *very* positive light.

Metrics Router

Depending on the size of your company or the complexity of your production architecture (or lack thereof), you may never find yourself needing anything resembling a metrics "Event Stream". This particular pattern is more common in large distributed systems with a tremendous volume of telemetry data and the need to support a variety of emitter and consumer interface formats.

It helps to think of this component as something akin to a metrics transport or router. In many cases, the need to publish a measurement and have multiple consumers pull copies of the same data is a primary driver for these services. Apache's Kafka (*http://kafka.apache.org/*), a "high-throughput distributed messaging system", is hugely popular for producing centralized feeds of telemetry data. Projects like Heroku's Logplex log router (*https://github.com/heroku/logplex*) and Mozilla's Heka stream processing service (*http://hekad.readthedocs.org/*) have different approaches but arguably address the same problem set.

Aggregation

There will be times when it's necessary to perform aggregation on your metrics stream. For example, application telemetry tends to be irregular, firing whenever an event or action occurs. This doesn't jibe well with the sequential and predictable nature of time-

series data. If our Whisper policy is designed for 10-second resolution, attempting to write a few increments in one interval and no metrics at all in the next interval is going to result in a loss of data in the former and gaps in the latter. Using aggregation, we can normalize our metrics: all of the data in the first interval should be summed up before transmitting, while the second interval should be reported as a zero (rather than nothing at all, resulting in a *null* value).

And although Graphite's render API provides all of the transformative functions you'll probably ever require, asking it to perform a lot of recalculations on the fly for numerous clients can result in significant CPU overhead in the Graphite web application. Many users prefer to apply statistical transforms in the aggregation layer (e.g. summing two metrics to create a new unique metric) before passing it on to the storage layer. Graphite's **carbon-aggregator** is an example of this type of service, and we'll cover it at greater length in the next chapter.

We've already touched on StatsD in terms of the library support it offers developers for easy instrumentation of their applications. But one of my favorite features is its ability to export "value-added" statistics for *timer* measurements. Not only will it tell you the average (*mean*) for all timers in an interval, it will also give you: the *upper* and *lower* boundaries, the total *count* of timers reported, the *timer_ps* of timers per second, the *sum* and *sum_squares*, and the *median* and *std* standard deviation. It's a veritable smorgasbord of statstical delights.

Last but not least, anyone with a reliance on logging data will want to look at aggregators like Logstash (*http://logstash.net/*) to parse the source logs and convert them into time-series data. Logstash even has support for StatsD output, or you can configure it to fire your metrics directly to Graphite's **carbon-cache**.

State Engine

In many ways, the state engine acts as the brains of a monitoring system. It may perform additional services, but it should at least: track the state of metrics according to their current and recent measurements in relation to defined thresholds (e.g. upper and lower boundaries), open alert incidents when a threshold condition is exceeded, close or resolve alert incidents when a threshold has recovered, and trigger alert or recovery notifications.

Despite all its warts, Nagios is an effective state engine. I have my own personal biases against the way it handles acknowledgements and flapping, but at its core it does a solid job adhering to the basic requirements for this component. Riemann (*http://riemann.io/*) and Sensu (*http://sensuapp.org/*) are both compelling alternatives to Nagios here, designed with automation in mind and the goal of streaming all check results (read: metrics) to a time-series storage engine like Graphite.

There are other Open Source alternatives that are more strictly aligned with the Composable Monitoring System design. Heroku's Umpire (*https://github.com/heroku/umpire*) is one of the more unique designs, specifically built to accept Graphite query definitions and thresholds. It then analyzes the Graphite response, determines whether the response falls within the acceptable threshold boundaries, and finally returns an HTTP status code 200 (success), 500 (out of range), or 404 (no data). This is a particularly ingenious design; with the combination of Umpire and any website monitoring service (e.g. Pingdom), it allows developers to build their own monitoring checks without needing to commit configuration changes to the monitoring system or burdening other IT teams with routine requests.

Self Service Monitoring

Umpire adheres to the notion of *Self Service Monitoring*. This is an approach to monitoring that aims to remove the traditional hurdles of deploying monitoring system configuration changes. There are certainly parallels to be made to concepts introduced by the *DevOps* movement, but the notion of Self Service Monitoring focuses squarely on the design of our tools in order to remove telemetry deployment friction for engineering teams.

Notification Routers

We've already discussed Alerting earlier in this chapter, so you should already be familiar with the basic concepts here. Notification engines are typically responsible for routing alerts to their desired destination, which means they should be capable of supporting a variety of output transports and protocols: Email, SMS, Webhooks or even other third-party "incident management" services.

Scheduling and escalation management is generally a difficult nut to crack, so there are unfortunately very few Open Source projects that address this problem directly. Companies like PagerDuty, OpsGenie and VictorOps do a very good job solving this concern, and in my opinion, are well worth the investment.

Storage Engine

Storage engines are responsible for long-term storage and retrieval of metrics. Due to the nature of time-series data, they need to support a heavy write load, but be suited for near-realtime retrieval. They should include transformative functions (standard arithmetic and statistical primitives) and support a variety of output formats: JSON, CSV, SVG and some manner of raw data should be considered standard fare these days.

Above all, they should be capable of persisting data to disk for long-term trending. It's not sufficient to store metrics in memory for faster response and then discard or allow

them to "fall off" after a predetermined interval (that's cheating!); we should be able to trust our storage engine to actually write our data to archive storage in a reliable manner.

If you've been paying attention so far (good on ya, mate), you'll probably find it as no great surprise that I recommend using Graphite as your storage engine. Thanks to its powerful API and scalability-conscious design, it's a natural fit as the *source of truth* for any monitoring architecture. Newer versions of Graphite include support for *pluggable storage backends* (such as Whisper), allowing users to choose the storage tradeoffs that matter for their particular use case.

Visualization

For many Graphite users, it might seem unusual to categorize Graphite as a "Storage Engine" but not necessarily as a visualization tool. It's always included support for PNG image output, so why not?

The world of data visualization has moved steadily away from server-side static images, adopting modern frameworks like D3.js (*http://d3js.org/*), built on standard technologies that incorporate Javascript, HTML and CSS. These frameworks support a much better interactive experience and offer a huge variety of dynamic new chart types that even systems like Graphite can't match. This separation of responsibility means that we now have an entire community of web designers and developers participating in the monitoring and visualization communities. If you've ever seen a Systems Administrator design a website, you understand this means a huge advance in usability (wink).

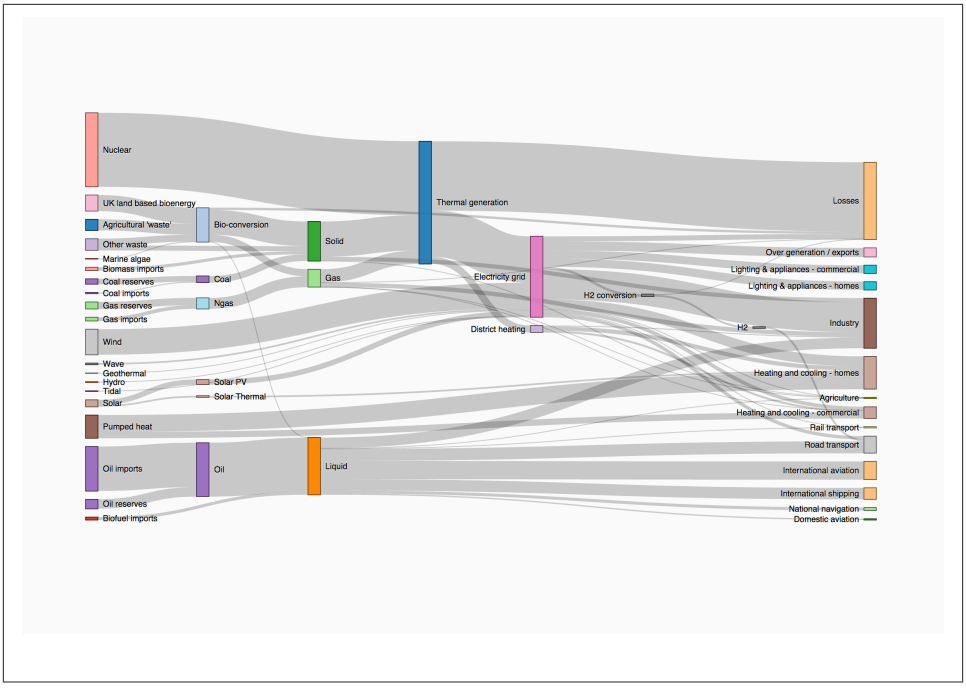


Figure 2-3. Sankey Chart

Dashboards in particular have benefitted from these changes. An enormous ecosystem of dashboard projects and commercial dashboard services now exist, able to consume storage engine APIs and generate useful interfaces for a variety of workflows and organizations. We'll cover some of these dashboards, as well as the client-side development approach, in Chapter 7.

As an aside, there are server-side benefits to decoupling image generation and the storage engine. Data requests from the client browser to the backend storage engine can take advantage of text formats like JSON, allowing for increased compression in transit, resulting in faster data connections and lower bandwidth requirements. Coincidentally, I'm proud to have been ahead of the curve here. I submitted the original patch for adding JSON format support to Graphite back in 2011. Fortunately for us all, the JSON formatting code that exists now bears no resemblence to my original patch.

Conclusion

As you can see, there are a *ton* of functional responsibilities for a modern monitoring system. Assembling yours (or even just buying one) will require a lot of considerations and almost certainly some compromises. I encourage you to take some time to analyze your current strengths and weaknesses, and to determine which gaps are important for

you to address *right now*. For most companies I talk with, time-series collection and visualization is a vital operational need for them, which is also probably why you're reading this book.

Now that you have a solid foundation of monitoring-related vocabulary and conventions, I feel confident that we can dive head-first into learning Graphite without too many distractions. The next chapter is rich with tips and tidbits that will help you get the most out of Graphite as it grows with your organization. There's a lot to absorb, but this knowledge should prepare you for many of the little surprises that trip up most new users (and even some experienced ones).

CHAPTER 3

Graphite Components: The Moving Parts

If you read the first two chapters, pat yourself on the back. There are a lot of technology considerations that go into a modern architecture, and I want readers of this book to understand many of the choices available before we move on to the Graphite nitty-gritty. I assume that anyone reading this book is probably going to be working on, or possibly even building their own monitoring system, so it's important to me that you're heading into this project with *eyes wide open*.

For readers who skipped ahead, you missed the free Starbucks coupon and a brief overview of the Graphite architecture. To bring you partly up to speed (minus the coffee, you blew it), there are three main "components" in Graphite: the Carbon daemons, the Whisper database library, and the Graphite web application.

This chapter will explain the role of each component as we begin to mentally "map out" our Graphite installation. It helps to understand how each piece fits together, especially among the different Carbon daemon types, so that when you you need to diagnose your Graphite server (or begin scaling it up beyond a single node) you'll have an appreciation of each component's behavior, performance characteristics, and potential bottlenecks.

Carbon

Whenever someone refers to Graphite's Carbon component, they're almost always talking about the `carbon-cache` service. New users may not even realize there are actually three distinct daemons collectively referred to as *Carbon*: the `carbon-cache` listener, `carbon-relay`, and `carbon-aggregator`. The latter two should be self-explanatory, but there are some subtle differences that make each appropriate for specific use cases or in combination as part of a complex architecture.

carbon-cache

The `carbon-cache` daemon is the hardest working process in time-series. It's written in Python and based on the Twisted library, an event-driven networking engine. But to describe it merely as a cache would be a major disservice (and not a particularly accurate one either). `carbon-cache` is a network service that accepts inbound metrics from clients, temporarily stores the metrics in memory (the "cache"), and then writes them to disk as Whisper database files. It accepts metrics over TCP or UDP as "plain text" or using Python's *pickle* protocol (an efficient serialization format). If you support a number of distributed systems, Graphite even supports the AMQP message bus. Because Graphite needs access to *both* persisted and in-memory metrics, the cache daemon also includes a query port that the Graphite web application polls for "hot" metrics that haven't yet been written to disk.

Solid State Drives

Because so much of its work is IO-related, `carbon-cache` is particularly sensitive to slow disks and low memory conditions; especially when the former causes the latter to balloon uncontrollably. For this reason I universally recommend solid-state drives (SSD) for all Carbon nodes. You can survive on traditional hard drives in a striped RAID configuration, but in the long term SSD is faster and more reliable.

Basic installations can usually get by with a single `carbon-cache` process. Each "cache" binds to its own CPU core, so it's easy to track down an overworked cache process with simple UNIX tools like `top` or `ps`. If history is any guide, your basic Graphite server will soon become popular enough that you'll hit the IO limits of that individual process. Fortunately, `carbon-cache` was designed to allow running multiple instances in parallel. And because each runs on an independent CPU core (assuming you have enough cores), you can keep adding cache processes as long as your disk subsystem can keep up with the aggregate writes.

You might be asking yourself how your clients connect to Carbon if you're running multiple `carbon-cache` processes. Good news, this is where `carbon-relay` enters the picture. Thanks to our next Carbon daemon, it's easy to add a single listener that "fans out" connections across a pool of cache instances.

carbon-relay

As the name implies, `carbon-relay` is designed to relay metrics from one Carbon process to another. You can think of it as the duct tape of Graphite systems. Anytime you need to grow your Graphite cluster, `carbon-relay` will be almost certainly be involved.

There are two primary tasks that the relay is designed for: forwarding metrics to another Carbon daemon, and replicating metrics to one or more destinations. This opens the door for a seemingly unlimited number of patterns and use cases where you need to add load-balancing or replication to your Graphite cluster. Depending on your situation you may find yourself using `carbon-relay` to scale up a single Graphite server with 16 `carbon-caches` running in parallel, to replicate metrics from one datacenter to another, or some combination of both.

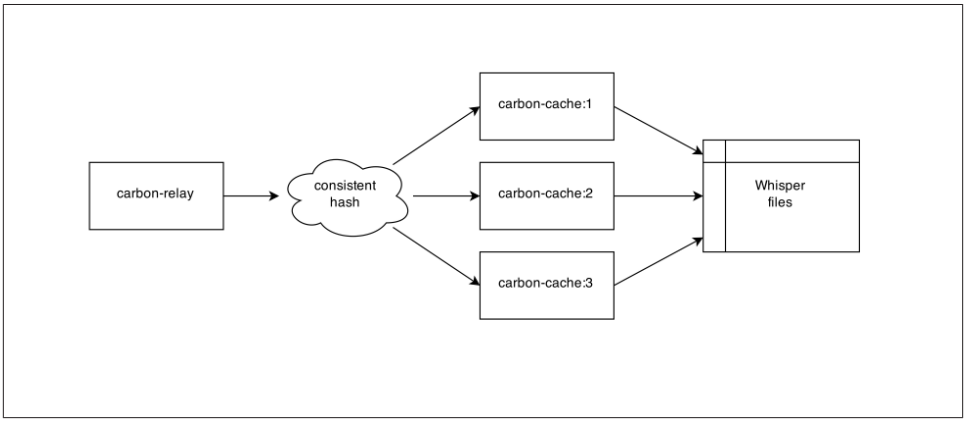


Figure 3-1. Basic carbon-relay arrangement

In some advanced configurations you'll even see folks using an external load-balancer such as HAProxy to distribute traffic across multiple `carbon-relay` instances. Just as `carbon-cache` is known to exhaust IO in busy Carbon servers, `carbon-relay` hits a similar limitation with CPU performance. Scaling out to multiple relays is an easy way to add capacity on a single node. Just watch out for long-running client connections; these won't balance properly across your backend pool since the connection isn't actually terminated on the relay itself.

We'll cover more advanced configurations in the later chapters. For now it's enough to understand some of the basic capabilities that Graphite and Carbon make possible.

Consistent-Hashing

The relay supports a few different routing algorithms for determining the next hop among an array of destinations. This is necessary to ensure affinity for a particular metric to a given destination every time. Without this affinity you could end up with a metric that has some fraction of its datapoints on server and the rest of an entirely different server. It's important to avoid this sort of split-brain delivery (at least with Whisper) because Graphite incorporates a "first-match" policy for any queries to Carbon nodes. In a worst-case scenario, Graphite would accept the response from a Whisper database

with incomplete data, drawing charts with gaps for your end users. For most users this isn't just ugly, it's unacceptable. Fortunately this is also completely avoidable with a little foresight and planning.

By default, `carbon-relay` uses the *consistent-hashing* routing method. This method hashes the metric string to ensure that it picks the same destination among a ring of nodes, every time. Consistent hashing aims for an even distribution of your metrics across all nodes. It's a very convenient approach if your storage isn't expected to scale too quickly.

When new nodes are introduced to a Graphite/Carbon/Whisper cluster (node ring), all of the datapoints for a fraction of your existing metric inventory (ideally `1/n` metrics, but not guaranteed) will need to be redistributed manually to the new node. This has traditionally been a major pain point for Graphite administrators and one of the drawbacks to the Whisper design. There are now tools like Carbonate available that make this data migration a much more palatable proposition, but it's still enough to make many users consider *rules*-based routing instead.

Rules-based Routing

In contrast to consistent-hashing, the *rules* relay method must be configured and maintained by the Graphite administrator. This method uses a dedicated configuration file to map destinations defined by literal strings or regular expression pattern matches. A rules-based setup will demand more ongoing maintenance, at least in terms of monitoring the storage use over time. Conversely, it will be far less painful to add nodes to your cluster if you get to choose precisely where your metrics are routed.

I like to think of the rules method as amortizing the pain of upgrading your Graphite cluster storage over time, versus inflicting all that pain on yourself at the time of the upgrade. I've seen large companies start with consistent-hashing, switch to rules after a particularly thorny upgrade cycle, and then back again once they had better synchronization tooling in place (similar to Carbonate). Regardless, it's best to make your choice and run with it. You'll never finish if you never get started.

Aggregated-Consistent-Hashing

There's one final relay method that was added to guarantee affinity when routing metrics across a pool of `carbon-aggregator` instances. It's especially important that metrics used in aggregation rules get handed off to the same aggregator. With the standard consistent-hashing algorithm, the next hop is based on a hash of the source metric. The *aggregated-consistent-hashing* algorithm instead hashes the intended aggregation path, ensuring that the source metric datapoints for each aggregation rule are sent to the same aggregator process.

If you intend to use rules-based routing, you can safely ignore this method. But if you're planning to use consistent-hashing for your metrics **and** you want to use `carbon-`

`aggregator` (behind relays) to aggregate some of your metric classes, you'll definitely need to employ this feature.

carbon-aggregator

Say there comes a time where one of your applications is emitting metrics at an irregular interval. You want to avoid gaps in your graphs (technically this isn't a problem with Graphite anyways thanks to the `transformNull` function, but humor me for a moment); ideally we should be able to buffer the metrics and then release them at routine intervals to match our retention policy.

In another scenario we might want to aggregate a collection of metrics into a single new metric. For example, what if we wanted to track the average latency across a series of application servers? The Graphite API will let us do this easily, but it could result in a significant performance penalty if you ask Graphite to render the average of these metrics across tens, hundreds, or almost certainly, thousands of servers all at once.

The `carbon-aggregator` addresses both of these situations. This daemon allows you to define rules that specify one or more source metrics (using wildcards), an aggregation method (sum or average), the amount of time (in seconds) to buffer before publishing the result, and the new metric name. We'll cover the specifics of the configuration in the next chapter, but here's a little taste to whet your appetite.

Example 3-1. Sample rules for carbon-aggregator

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

Generally speaking, this daemon will usually be found sandwiched between Carbon relays and caches, but that's not a strict requirement. In fact, `carbon-aggregator` shares a lot of code in common with `carbon-relay`, which means that it's also designed to hand off metrics to one or more caches on the backend. However, it does **not** have the same complement of routing methods as the relay. The aggregator only supports *consistent-hashing* to downstream destinations. Therefore, if you need to apply *rules*-based routing, you'll want to insert one or more relays between your aggregators and your caches.

Filtering Metrics

By now you should have a pretty good idea how each Carbon daemon processes metrics and interacts with the other daemon types. `carbon-cache` performs the heavy lifting, buffering datapoints in memory and routinely flushing them to disk. `carbon-relay` adds flexibility in terms of metrics routing, load-balancing, and replication. And finally, `carbon-aggregator` helps normalize our datapoints in a predictable manner.

One feature set that's common to *all* of the Carbon daemons is the ability to *whitelist* and *blacklist* metrics as they're received. Technically, the blacklisting happens before the

whitelisting, so it's important to define your rules accordingly. Filtering can be enabled on a per-instance basis for each daemon type (using the `USE_WHITELIST` setting), but the same ruleset is applied *across all instances* on the same server.

Beyond the common use case, I've found that one particularly effective use for the blacklist is to filter out source metrics after they've been aggregated by the `carbon-aggregator`. For example, you wouldn't want to keep a bunch of per-core CPU metrics hanging around after you aggregated them into a single stream. We can use blacklisting to match those per-core metrics and drop them on the floor before they make it to the filesystem.

But don't forget - the aggregator needs to process them first, so we can't activate the filtering at that stage. The blacklisting should be enabled at the `carbon-cache` (or `carbon-relay` if you're relaying between aggregators and caches), after the metric has already been "transformed" into the new aggregate metric.

Internal Statistics

One of the most important resources in your Graphite toolchest, not surprisingly, is the robust collection of internal activity and performance metrics. Each Carbon daemon tracks a variety of statistics relevant to their respective workload. You'll find some metrics that are common to all Carbon daemon types, such as CPU and memory footprint. There are also quite a few daemon-specific statistics, including the number of cache queries, cache overflow size, and "Whisper files created" metrics only reported by `carbon-cache` instances.

I can't stress enough the importance of this data when it comes to the care and feeding of your Carbon/Whisper nodes. A seasoned Graphite administrator should not only have a dashboard and charts showcasing these metrics, but will also have alerts set to fire on **both** upper and lower thresholds for many of them. We'll explain each of these in depth later in the book as we cover troubleshooting and debugging your Graphite cluster.

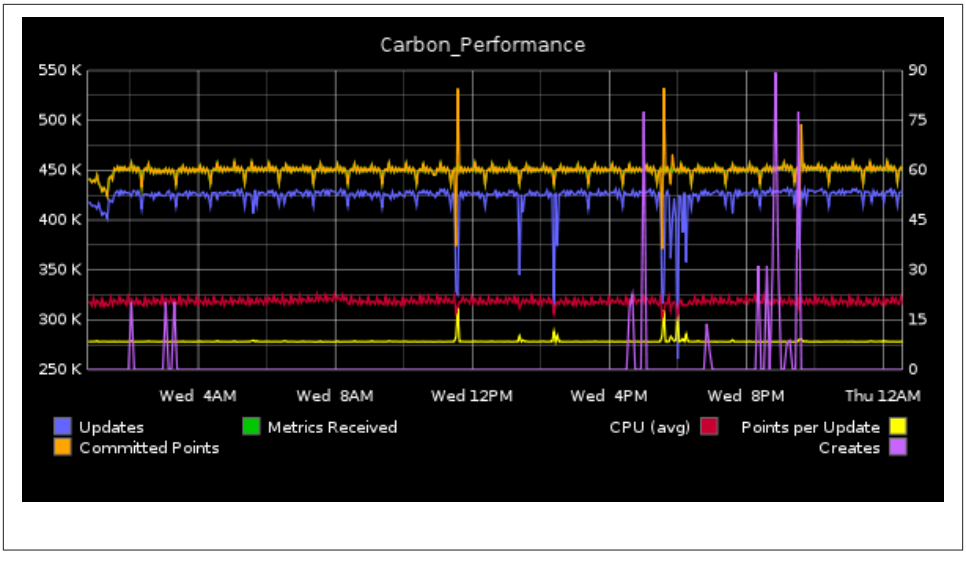


Figure 3-2. Monitoring the performance of carbon-cache

One word of caution - if we should ever meet and you ask me about scaling your Graphite server, be prepared to tell me how many metrics per second your cluster is writing. If I had a nickel for every time I've asked someone that question and their response was a quizzical shrug… well, let's just say it's still a monotonically increasing value.

Network Security Considerations

When it comes to getting metrics into your monitoring pipeline, there are few time-series systems that make it as easy as Carbon. Unfortunately, there's a big tradeoff here in terms of network security. Carbon has no support for encryption using any of the native transport mechanisms (TCP, UDP and pickle). And although the AMQP specification supports SSL and TLS, Carbon's listener doesn't use it.

Worse yet, it lacks any sort of authentication or authorization mechanisms to limit access or enforce namespacing, respectively. If a rogue client learns the location of your Carbon listener, they could easily overwhelm your system with bad metrics, or far worse, overwrite good data with bad.

My recommendation is to run your Carbon servers only on trusted networks. If you need to expose your listening port(s) to untrusted networks, make liberal use of firewalls and Access Control Lists (ACL). VPNs or IPSec should be used to encrypt data between datacenters and offices.

Whisper

Whisper is a fixed-size database *file format* with support for datapoints down to one-second precision. Every datapoint in a Whisper file is stored as a big-endian double-precision float with an epoch timestamp. In layman's terms, this just means that Whisper stores each datapoint with enough decimal points to guarantee a high degree of accuracy.

How do Whisper files get created?

Every metric you send to Carbon will map to a Whisper file on the underlying filesystem. Each dot-delimited segment of the metric name corresponds with one level of the directory path (relative to the Whisper storage root), with the final "leaf node" as the actual name of the Whisper database file. For example, a Whisper file stored at `/opt/graphite/storage/whisper/foobar.wsp` would represent the metric named `foobar` in a default Graphite installation.

If Carbon is unable to find an existing Whisper file that corresponds with a metric it will create a new Whisper database file to hold that metric datapoint and all future datapoints that arrive for that metric. It finds the proper retention configuration stored in the `storage-schemas.conf` file and uses those settings to create a new database file with the necessary metadata and archive definitons. Whisper then pre-populates the database with a null datapoint for each predetermined interval. These "empty" datapoints are replaced whenever a new datapoint is received for the corresponding timestamp.

Example 3-2. storage-schemas.conf

```
[collectd]
pattern = ^collectd\.
retentions = 10s:1w, 60s:1y

[default]
pattern = .*
retentions = 60s:1y
```

The use of these null datapoints might seem expensive from a storage capacity argument, and you'd be right. But this is also what allows Whisper to accept out-of-order datapoints and support historical backfilling so effectively.

Retention Policies and Archives

Whisper is designed to hold datapoints for a finite period of time. Theoretically you could choose to hold onto the original datapoints for 100 years, but each resulting file would be quite large relative to their actual usefulness. A single metric with 10-second precision stored for 100 years would require 370 megabytes (MB) of disk space.

To work around this limitation, Whisper allows you to store "copies" of the data at lower precisions. Each datapoint from a lower-precision archive represents the aggregated values from a higher-precision archive. In a typical scenario you might want to keep one-minute precision data for a couple weeks to support oncall and troubleshooting efforts, but then "rollup" those datapoints into five-minute precision for three years for reviewing seasonality and capacity planning efforts. This particular schema would only "cost" you 4MB a file. To contrast, storing one-minute precision for three years would be a 20MB database.

This might not sound like a huge difference, but now multiply that by a million metrics. Trust me when I say a million metrics *is totally feasible*. Most Graphite deployments become very popular once users see how easy it is to send metrics and visualize data. If you're not prepared for the explosion of "metric growth" your systems **will** be overwhelmed far sooner than you expected.

Calculating Whisper file sizes

So now you're probably asking yourself, "how **do** I prepare for the growth of my filesystem"? If you're a fan of 1980's cartoons, you know that *knowing is half the battle*. With a little math we can determine exactly how large any Whisper file will be according to its retention policy.

Every database has a 16-byte chunk of metadata at the beginning of the file. Each archive within the database has 12-bytes of embedded archive information. And each datapoint takes up 12-bytes of space. With this information we can quickly predict the file size for any Whisper database.

Say we have a new metric `foo` with a retention policy of one-minute precision for two weeks and a rollup to five-minute precision for three years (`1m:2w 5m:3y`, or `60:1209600 300:94608000`). The first archive has 20160 points (`1209600 / 60`), which we multiply by 12 (bytes per point) and then add 12 bytes for the per-archive info, for 241932 total archive bytes.

Next, we have a rollup archive with 315360 points (`94608000 / 300`), multiplied by 12, and then add our 12 bytes of info, for a total of 3784332 bytes.

Finally, we add the two archives with the 16-bytes of database metadata for a grand total of 4026280 bytes. Now let's test our calculations.

Example 3-3. Creating a Whisper file manually

```
$ whisper-create.py foo.wsp 1m:2w 5m:3y
Created: foo.wsp (4026280 bytes)
```

Yeah, that's a lot easier but it's not nearly as fun as math, am I right? As your parents might say, it builds character.

Deconstructing a Whisper file

Example 3-4 shows output from the `whisper-info.py` utility provided with the Whisper project. This is a handy way to determine the retention policy for any given Whisper database file. Frankly, this is the *only accurate way* to identify the schema for a Whisper file, since the schema configurations in `storage-schemas.conf` are only referenced when each respective database is created.

Each Whisper file includes a header section with metadata and aggregation policies (we'll touch on these in a moment). Following the header we find a series of archives: the first ("Archive 0") stores datapoints at 10-second precision for up to one day (86400 seconds); the second ("Archive 1") *downsamples* into 60-second precision for one year (31536000 seconds).

```
maxRetention: 157680000
xFilesFactor: 0.5
aggregationMethod: average
fileSize: 12718132

Archive 0
retention: 86400
secondsPerPoint: 10
points: 8640
size: 103680
offset: 52

Archive 1
retention: 31536000
secondsPerPoint: 60
points: 525600
size: 6307200
offset: 103732
```

Each lower-precision archive must be divisible by the next higher-precision archive. From our previous example, we have a 10-second precision archive rolling up into a 60-second archive. But that same 10-second precision archive could **not** roll up into a 25-second precision archive, since 25 is not cleanly divisible by 10.

Overthinking Your Retention Policies

It's a good idea to pick the highest precision storage schema that your circumstances will allow, and "run with it". Avoid the temptation to overthink your initial schema configurations. No matter how well you plan you're guaranteed to uncover some completely unpredictable use cases for your telemetry and instrumentation. Ship it now and address the edge use cases as they reveal themselves.

But remember - storing time-series data will always be a trade-off between performance, cost, and convenience. Understand your compromises but don't obsess over them.

Which archive handles my query?

One of the more confusing aspects of rollups for new users is understanding that when a query is performed against a Whisper database, it returns results **only** from the *single highest-precision archive that is able to process the query range in its entirety*. Say we have a storage schema that stores the original data at 10-second precision for one day, rolls it up into one-minute precision for one year, and also into five-minute precision for five years. If a user asks for all data from the last two days, it will return all results from the one-minute precision rollup archive.

The notion of retention boundaries is definitely ripe for confusion. Even after I've explained the preceding paragraph, you might assume these archives as having *sequential boundaries*. That is, when the first archive "ends" at one day, the next archive begins and resumes until its boundary at one year. This is not the case at all. In reality, all archives end at the current point-in-time ("now") and work backwards some length in time equal to their respective `retention` value.

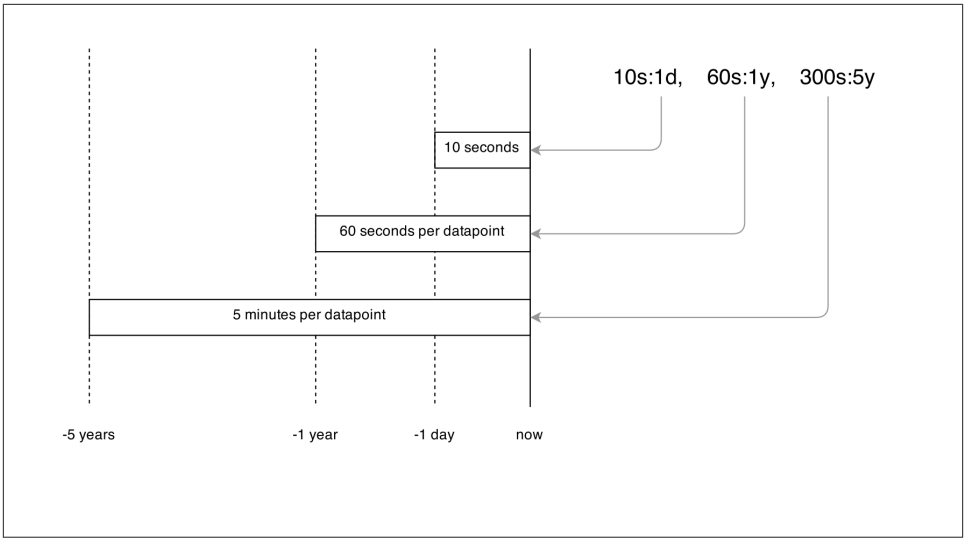


Figure 3-3. Whisper retention boundaries

Aggregation Methods

But how does Whisper actually "roll up" these datapoints into lower precision archives? I'm so glad you asked. Every metric has an *aggregation method* defined that tells Carbon which statistical method to use when aggregating a group of higher precision datapoints into a single lower-precision value. The default `aggregationMethod` is `average`.

Using our previous schema example (`10s:1d, 1m:1y`), let's examine how a sequence of datapoints would look after being rolled up with the **average** aggregation method. In Figure 3-4, the first column contains a series of 10-second precision measurements, each with an epoch timestamp and their respective value. The second column represents the aggregate value using the **average** rollup method.

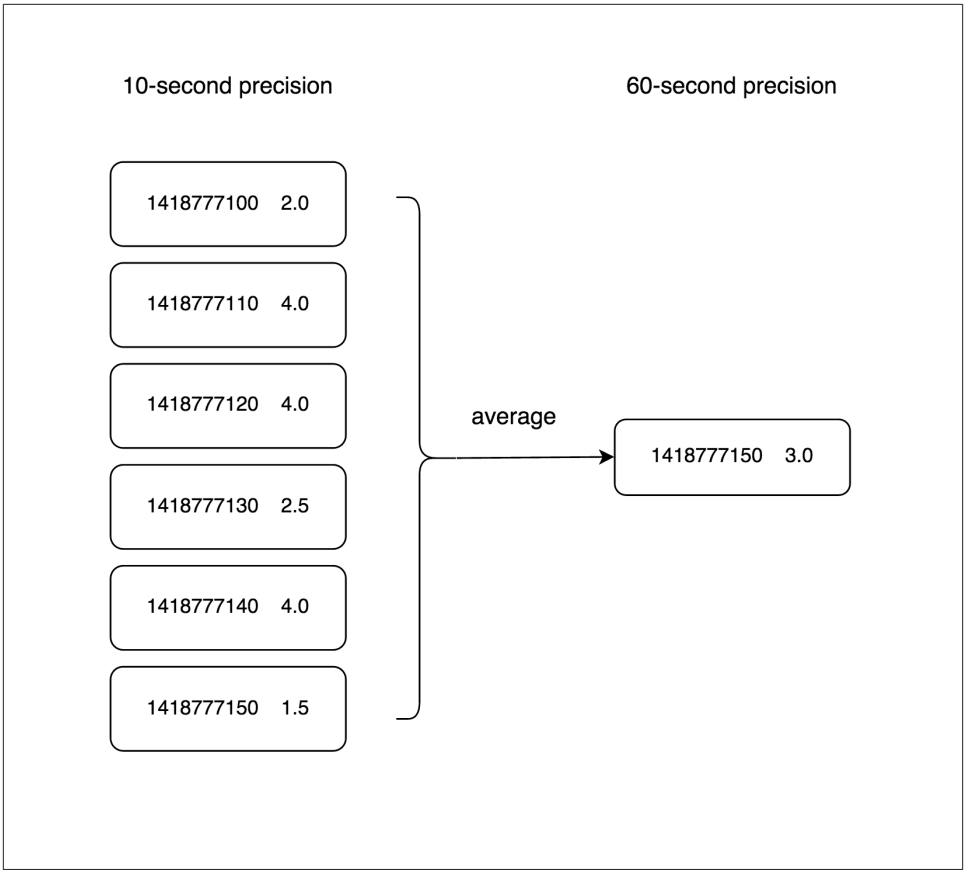


Figure 3-4. Aggregating rollups with average

Other valid settings for `aggregationMethod` include `sum`, `min`, `max`, and `last`. Because Whisper has no built-in comprehension of counters, gauges and timers, the onus falls to the Graphite administrator to pick the appropriate method for each metric. Although aggregating by average often makes sense for rates and gauges, you'll likely want to specify `max` for *increasing* counter values, `min` for *decreasing* counter values, and `sum` for counts (e.g. network interface octets).

It's important to understand the type of metrics that you're sending to Graphite and set the aggregation method accordingly, or you risk losing accuracy and "the long tail" of your data. The `aggregationMethod` and `xFilesFactor` settings are both configured in `storage-aggregation.conf` using the same pattern matching approach found in `storage-schemas.conf`.

Example 3-4. storage-aggregation.conf

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

[max]
pattern = \.max$
xFilesFactor = 0.1
aggregationMethod = max

[sum]
pattern = \.count$
xFilesFactor = 0
aggregationMethod = sum

[default_average]
pattern = .*
xFilesFactor = 0.5
aggregationMethod = average
```

xFilesFactor

Because Whisper uses precise intervals to define each retention level, it's not uncommon to see gaps in your data when something stops reporting metrics temporarily or reports erratically. We can adjust our *xFilesFactor* to accommodate these irregularities. This setting controls the proportion of metrics necessary to avoid downsampling into a *null* value.

Using our previous schema example again (`10s:1d, 1m:1y`), a default `xFilesFactor` value of `0.5` means that at least **50%** of your higher-precision datapoints *must exist* or the resulting aggregate will equal `None` (in Python-ese, generally considered *null*). As you can see in Example 3-5, we've specifically lowered `xFilesFactor` to `0.1` for min and max metrics, since we're more concerned with the existence of datapoints, rather than having enough to compute an accurage average. We've also effectively disabled `xFilesFactor` for counts, since we want to sum whatever is there. If nothing reports, this is acceptable and we should just return a zero.

Planning Your Namespaces

One of the most frequently asked questions I get revolves around *best practices* in terms of metrics naming and namespacing. Because Graphite doesn't support "tagging" or "labels" for multi-dimensional metrics, it's natural to embed, or encode, as much metadata as possible into the metric namespace. Metric names like `collectd.ord1.www3.disk-vda1.disk_time.write` and `app.skippy.srv2.controller.users.new.timer` are equally valid and include relevant descriptive elements that

help us identify the metric purpose. Nevertheless, there are some steps you can take to avoid future headaches.

Group by Collector
: It's a good idea to prefix each metric with the name of the collection agent, e.g. **collectd**, **diamond**, or even **my-perl-script-on-server-51**. This can be enormously helpful when you're trying to track down a misbehaving metric or sender.

Align your Metadata
: As much as possible, keep related metadata within the same depth of namespace. If your collectd metrics all report the datacenter name or region in the 2nd node of the metric name, try to do the same with your StatsD or custom metrics as well. It will make your life easier when you need to correlate metrics that may otherwise not group easily.

Whitelisting
: Mentioned earlier in this chapter, whitelists can help you tame the chaos that can happen when teams of developers are otherwise allowed to haphazardly fire metrics at your Graphite cluster. In particular, I like to use them to enforce good naming policies at the root of the namespace.

Provide a Metrics Sandbox
: One exception to the previous rule is to provide a test sandbox directory where *everyone* can send *anything* for the purpose of testing and experimentation. I usually create a `test` directory in the root namespace and make it clear that all metrics in that namespace will be deleted after a predetermined period (for me, usually two weeks). You'll need to set up a cronjob like the one in Example 3-6, but be *extra careful* not to delete production data outside the sandbox. You have been warned.

Example 3-5. Cleaning your metrics sandbox

```
$ crontab -l
0 4 * * * find /data/whisper/test/ -name \*.wsp -ctime +14 | xargs sudo rm
```

Performance Considerations

You might remember earlier when I emphasized using SSD for your Carbon/Whisper nodes. If not, allow me to reiterate: you should use SSD anywhere that Whisper files are being written. Whisper is an on-disk file format that needs to support a huge volume of continuous writes. You could probably survive on a large and fast RAID array, but the price of solid state drives have dropped far enough that it simply doesn't make sense to *not* use them.

Storing your Whisper data on a dedicated partition is a good idea if you're stuck on non-SSD drives, but it's a sound decision regardless in terms of capacity isolation. If you're not paying attention to your Carbon activity and your disk fills up, at least it won't affect the rest of the system logs and operations.

Finally, you'll want to keep an eye on the number of Whisper files you're creating and how that impacts your IO subsystem. Creating new Whisper files has a blocking effect on Whisper updates and reads, which can have a profound impact on your users' experience. If your updates can't flush to disk fast enough, they will consume the available memory on your server. If reads from Whisper (or Carbon) block or timeout, it can result in failed graph loads or gaps in your charts.

Graphite-Web

Whenever I hear that "Graphite is hard to install", there's a good chance they're talking about the Graphite web application (the unimaginatively named `graphite-web`). Because it's built on top of the Django application framework, it inherits a number of software dependencies. These are typically well-supported on the most popular Linux, BSD and Solaris operating systems and distributions. However, Python is almost excessively flexible in the way you can install modules and dependencies, which can lead to confusion and packaging conflicts.

In Chapter 4, we'll install the various Graphite components using a single, best-of-breed method that should be easy, repeatable, and work well regardless of your background or familiarity with Python. By avoiding the typical pitfalls we can focus on getting up and running quickly, and getting to the business of learning Graphite and having some fun with our data.

Nevertheless, we're still going to take a quick look at the various components that Graphite-Web *can* use. Some users will have a preference for Gunicorn (a Python WSGI server) over Apache with mod_wsgi, or MySQL over PostgreSQL, so it's good to touch on each of these to consider their relative strengths and weaknesses. There are a lot of component options available; I'm not going to walk you through the process of *installing* each different permutation, but we'll prepare you with the knowledge to make the best choice for *your* unique environment.

Django Framework

The Graphite web application is built using the Django framework. Django provides a lot of the boilerplate functionality common to most web applications: database interfaces, user accounts, administration tooling, etc. Unless you choose to contribute to Graphite development, you shouldn't need to think about the core Django bits much beyond installation and the initial setup.

Django has a couple direct dependencies that should be considered standard fare for any modern web application: a webserver to handle the requests and return responses, and a SQL database to store configurations and event data.

Webserver

The Django project recommends the Web Server Gateway Interface (WSGI) as the middleware service between your application and the webserver. There are a few popular choices for deploying Graphite using WSGI: Apache with mod_wsgi, nginx with Gunicorn, and nginx with uWSGI.

Unless you're dead set against running your Graphite under Apache, I recommend Apache with mod_wsgi. It performs well, has very good logging support (which you'll probably want for troubleshooting), and has a wealth of available modules for additional functionality. For example, if you want to use a centralized authentication service for your Graphite web application, you can lean on one of the many Apache authentication modules (although to be fair, Graphite has built-in support for LDAP).

If you're a fan of nginx, both uWSGI and Gunicorn are equally suitable WSGI servers. They're easy to install and configure, although you must proxy connections from nginx to the WSGI backend. My personal experience is that Gunicorn has inadequate logging, which is a back-breaker when you're trying to diagnose rendering issues in your Graphite cluster. However, both are popular WSGI choices and easy to swap in, so feel free to experiment to find the best fit for your installation.

Database

Graphite uses a relational database to store user profiles, user and group permissions, graph and dashboard configurations, and *Events* data (we'll touch on these later). Thanks to Django's Object Relational Mapper (ORM, used to abstract object models to the underlying data schema), it's relatively painless to get a database installed and running for our purposes. Using the ORM also gives us a manner of flexibility to pick the SQL engine that best suits our personal preference.

SQLite

By default, Graphite uses the popular *SQLite* database; it's fast, lightweight, easy to install, and supports all of the functionality we need for a basic Graphite setup. Best of all, SQLite is a file-based database format using a library driver (coincidentally, very similar to Whisper in this sense), which means there are also no additional network or firewalling considerations. It's a very good choice for a development or test environment.

However, SQLite is a *very* poor choice for production environments. Because SQLite is file-based, it uses write locks to control access to the database. What this means is that during writes to the database, SQLite performs a `fcntl()` on the file, typically for only a few milliseconds, before releasing it for other concurrent requests. Why is this a problem? Typically it's not. But if your SQLite file exists on a filesystem with heavy I/O operations (ahem, like a Carbon/Whisper node), and/or has poor I/O performance, you're gonna have a bad time.

Since our Graphite installation in the next chapter is basically a sandbox, we're going to use SQLite for the database. But I'm not a complete madman, so later on in the *Scaling Graphite* chapter I'll teach you how to migrate from SQLite to a real PostgreSQL database.

PostgreSQL vs MySQL

If you're serious about your Graphite cluster, you absolutely need to use a real RDBMS (Relational Database Management System) such as PostgreSQL or MySQL. Without getting into a religious argument as to which database is better, I feel I can make a case as to why PostgreSQL *is better for Graphite*.

For a long while now, PostgreSQL has been the preferred database among Django developers. Much of this stems from various technical arguments in PostgreSQL's favor, but the one that seems to matter the most is PostgreSQL's complete support for full-text indexing and searching. This is less relevant for Graphite than it might another web application built on top of the Django framework, but the net result is the same. Within the Django and Graphite projects, PostgreSQL-related features and bugs tend to get resolved quicker than their MySQL equivalents (at least from my own anecdotal experiences).

But if your business has a lot of experience running MySQL, you'll probably do just fine running Graphite on it as well. From my perspective, the benefits of knowing how to operate and maintain a healthy database far outweigh the slight performance increase you'd get by running it on an unfamiliar system.

Memcached

Possibly the easiest performance tweak you can make for Graphite is to install *Memcached* on your Graphite-Web node. Memcached is an in-memory key-value store that can be used by Graphite's web application to cache query results from the backend. This results in much faster render times for the user, as well as decreased load for the Whisper server and any `carbon-cache` daemons.

The only time you might run into issues with Memcached is in a larger Graphite cluster with multiple nodes and a dedicated web frontend, or "render pool". These web instances should be configured to use the same `memcached` service. You do *not* want to run independent `memcached` processes for each web frontend - **or** - on any of your backend nodes. Doing so can result in inconsistent data, leading to bad graphs and confused users.

A good general rule of thumb is to run a single Memcached, as close to your users (in the logical sense) as possible.

Events

Earlier in this chapter I gave a lengthy overview of the Whisper database format and the type of data it can store (hint: one-dimensional time-series values). In order to associate additional dimensions (e.g. location or role), that metadata had to be encoded within the metric namespace. Although the Graphite render API provides a number of different functions to circumvent this limitation, there are still times when *multi-dimensional* metrics would be much more suitable; for example, tracking a release version or user id whenever a code change is pushed to production.

The concept of *Events* was added back in Graphite version 0.9.9 to address this shortcoming. The feature supports tagging and attaching data "payloads" to each associated event. Although the render API is still limited as to how much of this data can be leveraged within an actual graph, it *does* allow you to filter events by their tags and render the frequency of the matching results.

Events are particularly useful for tracking commits, software releases, and application exceptions or backtraces. Generally, anything that you'd want to annotate as an isolated incident or a series of occurrences - where the value lies in their frequency or existence, *rather than the value associated with them*, is a good candidate for tracking as an event.

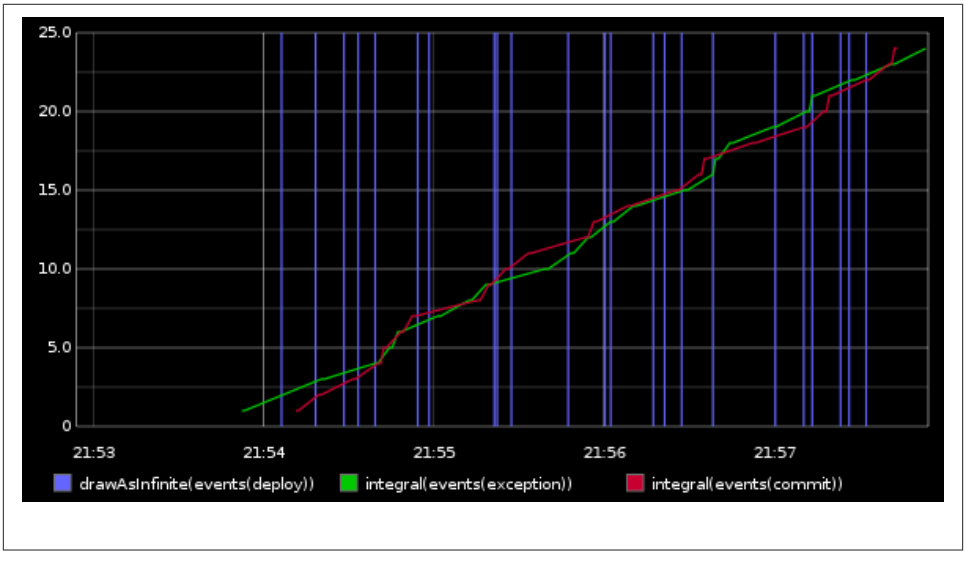


Figure 3-5. Visualizing Events

Unlike the usual metrics fed to Carbon and written out to Whisper files, events can either be submitted to the Graphite events API or entered manually through the Django administration UI. They are saved to the same relational database used by the web application for storing user profiles and graph & dashboard configurations. If you intend

to make judicious use of events, I again strongly encourage you to use either PostgreSQL or MySQL for your RDBMS.

Storage Backends

Before the web application can construct graphs (or simply return formatted data to clients as JSON), it must first pull the time-series data from the backend storage service. As we discovered earlier, Graphite stores its metrics in Whisper database files, with the ability to also query the `carbon-cache` query port for "hot data" that hasn't yet been persisted to disk.

In fact, every Carbon/Whisper node in your network **must** have a corresponding Graphite-Web instance running locally. This component acts as the "network interface" for any Whisper metrics stored on that host, responding to API calls from other Graphite-Web peers. In this sense, the clustering aspect of Graphite is a bit unusual: Carbon/Whisper nodes must have a Graphite-Web process to *read* metrics, but you may also find dedicated Graphite-Web "render nodes" which only serve as front-end web services, optimized for pulling results from their backend peers, and then transforming and rendering the results for users.

As it turns out, Whisper isn't the only storage format that Graphite-Web can read data from. Graphite has included support for Round-Robin Database (RRD) files since the very beginning. Developers have also added support for *pluggable storage backends* to the development branch of Graphite-Web. We'll touch on this briefly but our focus will remain on features found in the stable `0.9.x` release cycle.

RRD

Graphite is able to read from Round-Robin Database (RRD) files just as it does with Whisper. You may already be running some applications that write metrics out to RRD, such as Nagios, Cacti or Ganglia. As long as the web process running the Graphite web application has access to find and read the RRD files from the filesystem, it can use those metrics just as it would from metrics stored in the native Whisper files. There is no other special configuration or preparation necessary.

However, a word of caution against colocating your RRD files on a busy Whisper server. The performance characteristics of writing heavy IO to RRD varies significantly enough to warrant isolating the two database types. Just so we're clear, I'm not suggesting this because it's hard to find powerful systems with plenty of IO capacity. Nor is this the "because I said so" defense, commonly employed by my parents when I refused to clean my room. This is because mixing the two pieces of software on the same host becomes a *major pain in the arse* to diagnose when something goes wrong.

On a moderately busy system, you're probably ok to store the RRDs on a separate disk partition and symlink the RRD directory within the Graphite storage root (we'll cover

this in the next chapter). But if you expect a lot of writes to your RRD files, I would advise storing them on a completely separate system with its own dedicated Graphite-Web front-end. You can then treat this host as a backend cluster node (which we'll cover later in the *Scaling Graphite* chapter, I promise).

Ceres

Ceres is a "next-generation" time-series database format designed to replace Whisper as the default storage backend for Graphite. At least, it was when it was released over three years ago. In defense of Ceres, it's a good redesign of the RRD-style database format. Unfortunately, it has had to wait patiently for the next major release as maintainers have continued to roll out minor maintenance releases in the 0.9.x branch. However, I believe that Ceres will be a good upgrade for many users wanting to maintain their existing architectures rather than migrate to something based on a NoSQL backend, such as Cyanite (see below) or even to another system like OpenTSDB.

Unlike Whisper, Ceres is not a fixed-size database. It supports sparse files with relative timestamps; it doesn't require padding with null datapoints to pre-populate files with fixed-size resolutions. This results in significant storage savings and the ability to distribute any metric across multiple nodes. Graphite is able to query multiple remote Ceres nodes and coalesce the results; contrast that to Whisper, where the "first match wins" results in inconsistent data and graphs.

If you're interested in trying out Ceres, I urge you to try out the development branch of the Graphite-Web, Carbon, and Ceres projects. But be prepared that only a fraction of the Graphite community is running Ceres in production, so you aren't likely to see the same level of support (including from this book's author) that you will for Whisper.

Cyanite

One of the more exciting changes in the development branch of Graphite-Web has been the refactoring of the code that queries metrics from the backend storage system. This introduced a new setting known as `STORAGE_FINDERS`, which makes it possible to interface with (and design new) alternative storage layers for Graphite. The first such storage backend consists of the Cyanite and Graphite-Cyanite projects, developed by Pierre-Yves Ritschard and Bruno Renié, respectively.

Cyanite is a Cassandra-backed time-series database designed to be API-compatible with the rest of the Graphite stack. It provides a Carbon-compatible listener which receives metric and writes them out to the Cassandra schema. The Graphite-Cyanite project acts as a shim between the Graphite-Web `STORAGE_FINDER` interface and Cyanite's own HTTP API.

Like Ceres, if you wish to give Cyanite a spin, you'll need to run the development branch of the Graphite-Web project. Unless you already have experience running Cassandra in

production, this might not be the best storage option for you, but it certainly offers some upside in removing Carbon and Whisper from your potential list of scalability concerns.

Putting it all together

By now you should have a pretty solid understanding of each Graphite component. They each have their own performance considerations and bottlenecks; hopefully I've equipped you with enough knowledge to start thinking about your own configuration as we prepare to install Graphite in the next chapter.

Even with all of the detail I've brain-dumped on you over the first three chapters, I know a lot of readers absorb this material better in a visual manner. Therefore I've put together a collection of diagrams covering some common Graphite deployment strategies, particularly as they pertain to scaling Carbon and Whisper performance. If there was any confusion as to how these daemons fit together to "get the job done", I hope these examples will clear things up for you.

Basic Setup

Here we have a single `carbon-cache` listener accepting metrics and writing them out to local Whisper files. Users will visit the Graphite web interface, which in turn queries data from the `carbon-cache` and Whisper, coalesces the results, and renders a graph (or the requested output format). All of the Graphite components exist on a single physical or virtual server.

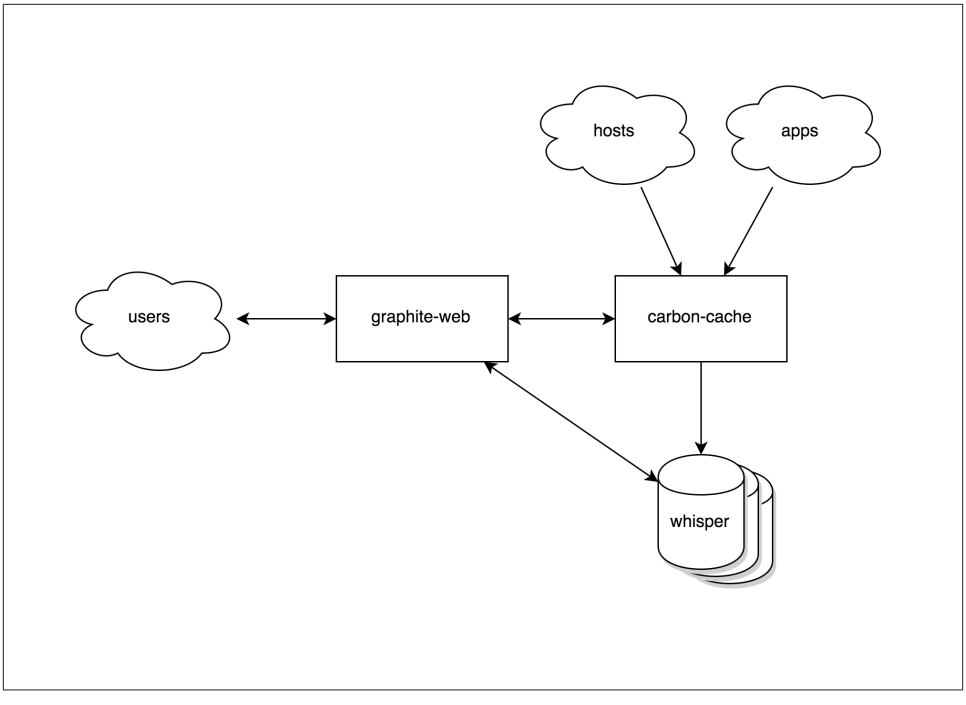


Figure 3-6. Basic Setup

There is no `carbon-relay` in this example because we're only dealing with a single local `carbon-cache` instance. We've also excluded any `carbon-aggregator` instances for simplicity; these daemons are in use at plenty of Graphite shops, but are not what I'd consider a frequent occurrence. We'll include them in some of our examples throughout the book, but usually only when it helps to drive home a specific use case.

Vertical Scaling

The term *Vertical Scaling* is generally used to describe situations where existiing hardware capacity is upgraded to increase the performance ceiling. Here, we're appropriating the term to describe our ability to squeeze as much "juice" as possible out of the existing (hopefully, appropriately specced) hardware.

Thanks to Carbon's design, we can distribute load across many cores on a single server. However, this approach can be (and frankly, should be) applied to *any* Graphite scaling plans. It doesn't make sense to scale out horizontally to multiple nodes if you're not also going to take full advantage of the resources within each individual node.

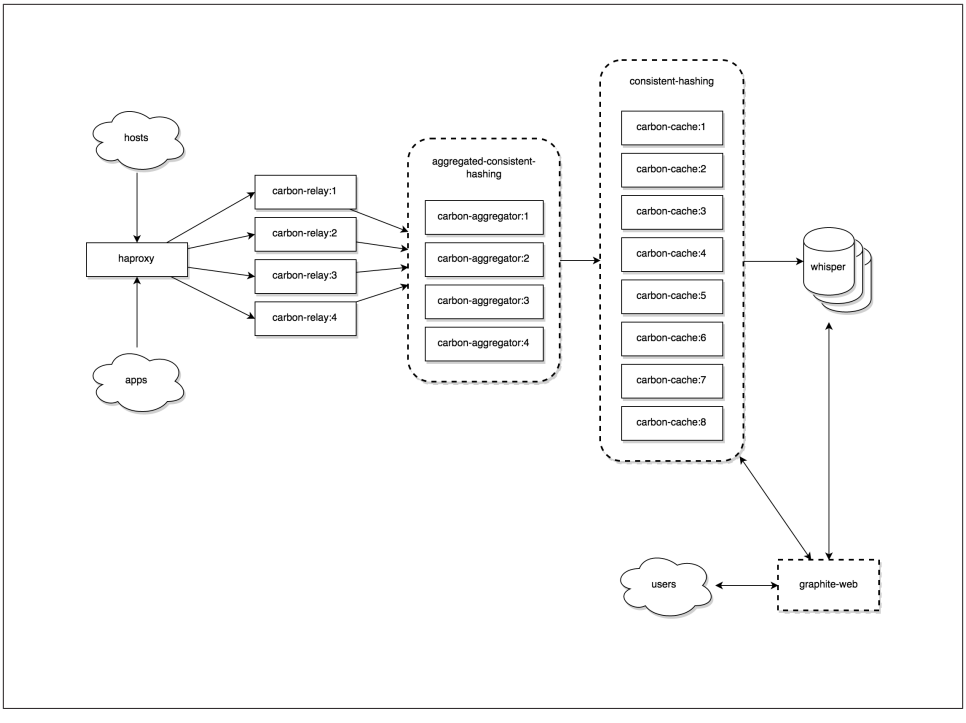


Figure 3-7. Vertical Scaling

In this example, all entities (except for the "users" and "app" & "host" metrics) exist on the same server. Metrics are fed into an HAProxy service which may exist on the same host (but isn't required). HAProxy performs a round-robin distribution across a pool of `carbon-relay` processes.

The relays in turn route their metrics to one of many `carbon-aggregator` processes using the *aggregated-consistent-hashing* distribution algorithm. This guarantees that all metrics related to any specific *aggregated metric* are sent to the same aggregator instance. As you may recall, this is necessary to ensure that aggregate metrics are calculated accurately.

All metrics are then sent on to one of eight parallel `carbon-cache` daemons using the *consistent-hashing* algorithm. This ensures a roughly equivalent workload across all cache processes. Lastly, the metrics are written out to Whisper files on the same filesystem. The Graphite web application provides the usual interface to all metric data stored on the server.

If you're counting on your fingers, you already figured out that we would need at least 16 cores to satisfy this configuration (also, you were probably counting on your toes). This doesn't include any dedicated cores for HAProxy or the base operating system

processes, so you'd probably want at least 24 cores on a single server for this particular example. Fortunately, it's easy to adjust the configuration to add or remove Carbon instances to fit your particular hardware specification.

Horizontal Scaling

Scaling out to multiple servers is a popular method for adding capacity to services that support it, and Graphite is no exception here. However, there is no built-in automated service for synchronizing metrics across node; therefore, it's very important to consider your *Disaster Recovery* strategy as you design your Graphite cluster.

Graphite's native mechanisms for scaling out horizontally can be leveraged to address *high availability*, write and/or read performance, or a combination of both. It would be impossible to cover every possible permutation or scenario, so instead we'll take a brief look at how Carbon's `REPLICATION_FACTOR` influences the utilization of a cluster of systems.

For this example we have a cluster of three servers dedicated to Carbon and Whisper storage. Each node has multiple `carbon-cache` processes and at least one `carbon-relay` process, allowing us to achieve greater performance using what we learned through *Vertical Scaling*.

In front of the cluster, an HAProxy instance is load-balancing metrics traffic to one or more `carbon-relay` instances. These relays may use **either** *rules* or *consistent-hashing* to distribute metrics across the members of our cluster.

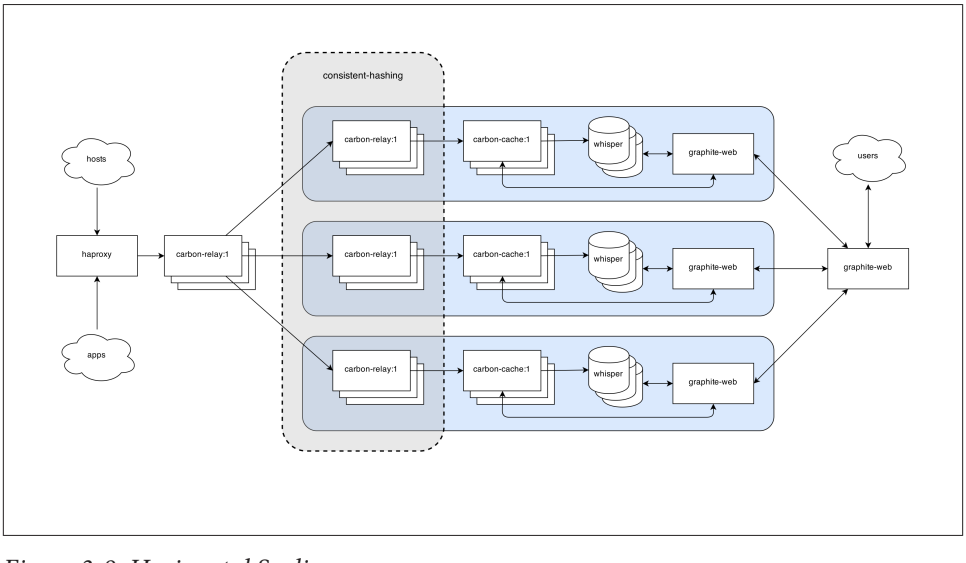


Figure 3-8. Horizontal Scaling

At this point, with zero replication, we've achieved some basic measure of load-balancing distribution across our cluster. We should see an approximate 200% performance boost compared to a single node. If one of the cluster members were to go down, clients would not be able to view data for any of the affected metrics on the inaccessible node.

To add some level of redundancy, we can adjust our `REPLICATION_FACTOR` (consistent-hashing only) setting for the external `carbon-relay`(s). By default, this is set to a value of one (1), meaning each metric will be routed to a single `DESTINATION`. Setting this to a value greater than one but less than the total number of nodes will result in partial replication; we would still achieve some performance gains but less than we would with zero replication. We could survive a single node failure without entering a degraded state, at least from the users' perspective.

With `REPLICATION_FACTOR` set to a value equal to the size of our cluster we achieve full replication. At this level we can lose all but one of our cluster members and still have a "working" Graphite installation. However, we have no performance gains at this replication level. As you can see, horizontal scalability offers a wide spectrum of choices (and compromises) to consider.

Multi-Site Replication

As a Graphite administrator, one of the big eye-openers for me was when I realized the power of using multiple layers of relays, each for a different purpose. Our final example demonstrates how we used this technique at a major DNS and Internet Services Provider to avoid a single point of failure and improve the user experience based on proximity to the closest datacenter.

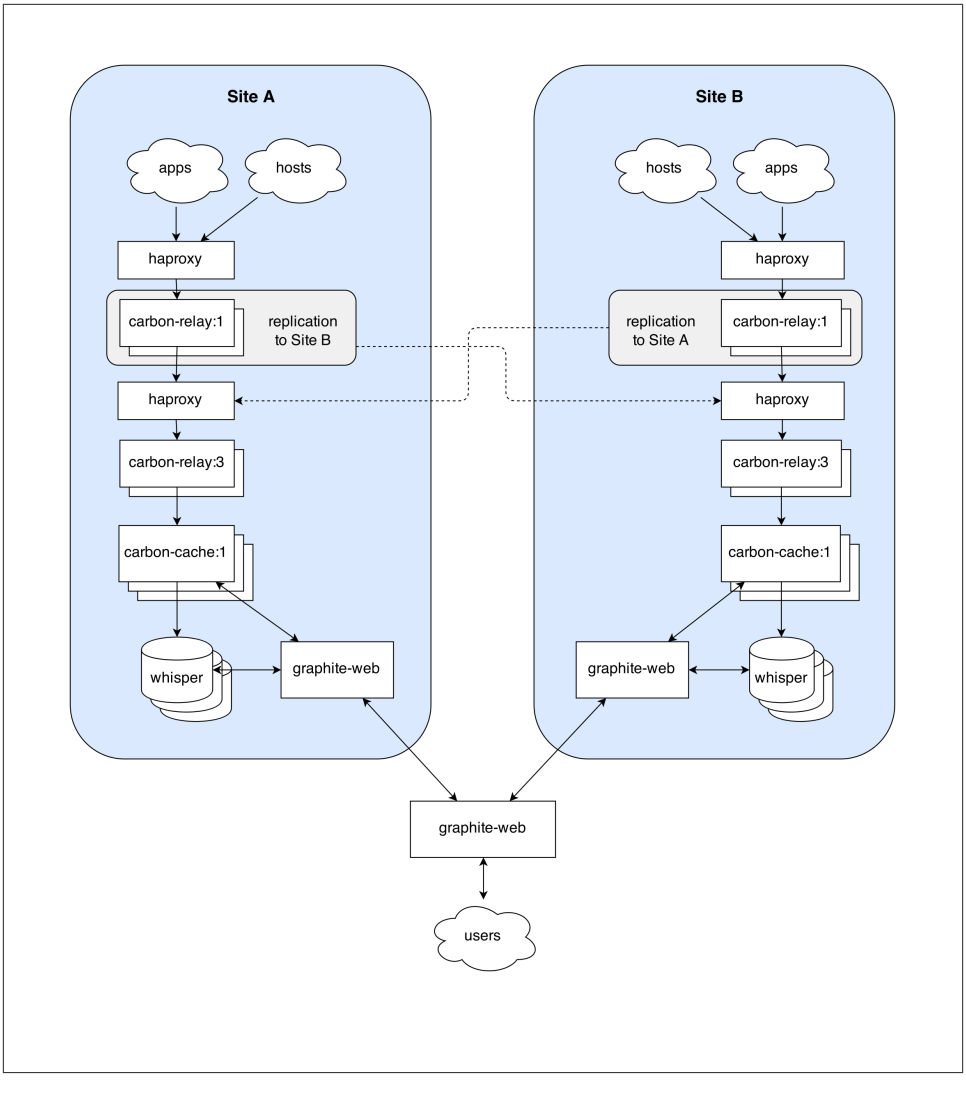


Figure 3-9. Multi-Site Replication

Being an aforementioned vendor of "really important internet services", it was vital that we had reliable (and fast) access to our production metrics. We had numerous datacenters spread across the globe, which meant our collection systems needed to be close to the systems being measured *and* to the clients reading the graph data.

We designed an architecture where our two largest "core" sites would host a Graphite cluster accepting metrics for the remaining sites. Using Anycast DNS, the systems in each "satellite" datacenter would send their metrics to an HAProxy instance running at

the nearest core site. HAProxy would in turn round-robin the metrics to the first layer of `carbon-relay` processes. This pool of relays was responsible for *replication*, sending a copy of all metrics to the next routing layer, an HAProxy service at the local site *and the **peer** core site*.

At this point in the stack, both core sites had a full feed of metrics from every datacenter throughout the world. Metrics were then load-balanced to another pool of `carbon-relay` processes, which would then distribute them across the pool of local `carbon-cache` daemons.

A Final Thought

One aspect I failed to touch on in the previous examples was the location of Graphite's web database in relation to the rest of the stack. Assuming that all of your components reside in the same datacenter, this shouldn't be an issue in most cases. However, in the previous *Multi-Site* example, we ran a PostgreSQL primary and secondary database configuration (the former in one core site, the latter in the other). Because of the round-trip involved in loading graph and dashboard configurations from the web database, it actually mattered which database our Graphite "frontend" web application was configured to query.

I mention this not because I expect this to be a common circumstance for many of you. Rather, I hope it helps to underscore the importance of good telemetry when scaling a complex system like Graphite. You can't poke around blindly at your Graphite cluster (or any piece of software) and expect it to fix itself. Well, you *can*, but you won't be very effective at your job or assisting others with theirs.

Garbage In, Garbage Out.

CHAPTER 4

Building Your First Graphite Server

Hopefully I haven't scared you away with all of the mental preparation and cautionary mumbo-jumbo. Graphite really is a fun and effective tool. But like any complex piece of machinery (or software), it takes an experienced operator to configure and maintain it for running at peak efficiency. The previous chapters should have imbued you with enough practical knowledge around the Graphite ecosystem to help you clear most hurdles head-on.

This chapter (finally) explains the recommended procedure for installing and configuring your own Linux instance running Graphite, Carbon, Whisper, and the database where we'll save our graphs and dashboards later. We'll also cover a variety of housekeeping tasks, including web access controls (authentication and authorization), fixing permissions for the storage and logging directories, and finally, testing that everything works as expected.

Note that you should be able to run all of the Graphite components on any modern UNIX or UNIX-like system, including Linux, Solaris, and the family of BSD operating systems. We only cover Linux since that is the clear leader in terms of numbers of Graphite installations, and to keep this chapter from becoming even more unwieldy than it already is.

If you're the kind of person who prefers to learn by diving in with both feet, I have good news for you. The Synthesize project was designed to help users quickly and painlessly launch a Graphite instance. We'll cover that early in the next section, so feel free to jump ahead and we'll meet up again later.

Installation

Are there containers or images already available?

Wow, it's like you're reading my mind. There's a good chance that you're already familiar with virtualization (and containerization) tools like Vagrant and Docker. While the latter is an enormously popular tool for deploying software using Linux Containers (LXC), it's rather a pain to use with Graphite due to the quantity and variety of executable processes and interconnected network services.

Fortunately, Vagrant fits the bill nicely, allowing us to build and share a virtual system configuration with our choice of Linux distribution, forward ports for host-to-guest communication (StatsD, Carbon, and Graphite-Web services), and include a provisioning script (for the actual installation process). This is the formula I chose for Synthesize, an Open Source project aimed at beginners who want a simple way to get up and running with Graphite.

Running Synthesize in Vagrant

Launching a Graphite virtual machine via Synthesize is just a few easy steps. The following assumes you already have Git (*http://git-scm.com/*) and Vagrant (*https://www.vagrantup.com/*) installed on your personal laptop or workstation:

```
$ git clone https://github.com/obfuscurity/synthesize.git
$ cd synthesize
$ vagrant up
```

Running Synthesize Manually

If you'd rather use Synthesize to install Graphite on a "real" system (i.e. not a VM running on your laptop), you can do that too. The only limitation is that it must be running the supported base operating system required by Synthesize. At the time of this writing, that's Ubuntu 14.04 LTS.

```
$ ssh server
$ git clone https://github.com/obfuscurity/synthesize.git
$ cd synthesize
$ sudo ./install
```

Next Steps with Synthesize

Once Synthesize is running, you'll have a finely tuned Graphite instance running the various Graphite components, a high-performance StatsD server implementation (Statsite), and collectd. The collectd daemon has been specially configured to gather a variety of health and performance metrics for the Graphite and Carbon services on the instance.

For a full breakdown of the various service ports, web credentials, uninstallation methods and more, I urge you to visit the project website (*https://github.com/obfuscurity/synthesize*) for the most up-to-date version of this information.

If you chose the *shortest path* with Synthesize, I won't be offended if you decide to jump ahead to the next chapter and start learning how to create graphs. On the other hand, you're welcome to hang around and see how the sausage is made, so to speak.

Where does Graphite store all my files?

No matter which method you choose, Graphite installs all files in the `/opt/graphite` prefix by default. Distributions that package Graphite for their users often choose a directory layout that's compliant with the Filesystem Hierarchy Standard (FHS).

I respect your choice to put your files where it makes sense for you and your business, but for my money I prefer my files in the default location. It's easier to back everything up in one fell swoop, and you know that the Graphite community will be better informed to help you out in case something goes sideways.

Nevertheless, many of the subdirectories can be changed in your configuration settings. Here are some of the more important directories and a brief description of what they contain.

`GRAPHITE_ROOT`
: The installation root for Graphite. Defaults to `/opt/graphite`.

`CONF_DIR`
: All Carbon-related configuration files are stored here. It's common to find some Graphite-Web settings files here as well (`dashboard.conf`, `graphTemplates.conf`, etc) although those can be overridden in Graphite's `local_settings.py` file (discussed later). Inherits from `GRAPHITE_CONF_DIR` if that environment variable is set. Defaults to `GRAPHITE_ROOT/conf`.

`STORAGE_DIR`
: The root for all storage-related files, including Whisper and RRD databases, logs and pidfiles. Inherits from `GRAPHITE_STORAGE_DIR` if that environment variable is set. Defaults to `/opt/graphite/storage`.

`LOCAL_DATA_DIR`
: Tells Carbon where to write your Whisper files. Defaults to `STORAGE_DIR/whisper`.

`WHISPER_DIR`
: Not to be confused with `LOCAL_DATA_DIR` (although they should be the same value), this setting tells Graphite-Web where to find all the Whisper files to read from. Defaults to `/opt/graphite/storage/whisper`.

`RRD_DIR`
: Where Graphite-Web should look for RRD database files. Defaults to `/opt/graphite/storage/rrd`.

`LOG_DIR`
: The directory containing logs for each of the Carbon daemons and the Graphite web application (`webapp`). Defaults to `STORAGE_DIR/log`.

`PID_DIR`
: The directory where Carbon daemons write their pidfiles. Defaults to `STORAGE_DIR`.

Example 4-1. Typical Graphite directory structure

```
/opt/graphite
├── /opt/graphite/bin
├── /opt/graphite/conf
├── /opt/graphite/lib
├── /opt/graphite/storage
│   ├── /opt/graphite/storage/log
│   ├── /opt/graphite/storage/rrd
│   └── /opt/graphite/storage/whisper
└── /opt/graphite/webapp
    ├── /opt/graphite/webapp/content
    └── /opt/graphite/webapp/graphite
```

Are packages available for my distro?

If you're using one of the more popular Linux distributions, there's a good chance that they're already building binary packages of the various Graphite components. Although you're certainly welcome to use those, I urge you to at least skim through the rest of this section so you have a better understanding of Graphite's software dependencies and installation process.

What installation methods are available?

The Graphite project officially supports two different installation methods: binary packages hosted on the Python Package Index (PyPI, found at *https://pypi.python.org/pypi*) and installed using the *pip* package manager, and manual installation from the project *source code* using the respective *setup.py* script for each component.

You might think that pip packages are the way to go, and for many folks they're a good choice. However, I personally prefer installing from source due to the flexibility you gain. Not in terms of installations or upgrades, so much, but rather for the convenience of testing new branches and bugfixes. In addition, having a local checkout makes it that much easier to contribute new Graphite-Web rendering functions back upstream. If you don't believe me, take a look at the list of contributors for Graphite-Web's `functions.py` sometime.

```
$ git shortlog -s webapp/graphite/render/functions.py | wc -l
63
```

Because I'm the one writing this book, I get the pleasure of informing you that we're going with the *source* installation method. Never fear, it's actually very simple and straightforward. The software dependencies are the same either way, so the difference between *pip* and *source* installs is minimal.

Should I use Virtualenv?

Virtualenv is a tool for creating isolated Python environments. It's commonly used by Python developers to build small, self-contained environments within each project's development folder. When you use (and *activate*) virtualenv, all Python binaries and libraries needed for that project are copied into the designated folder structure. This self-containment makes it possible to run countless different versions of the Python interpreter and 3rd party libraries on the same system, each safe from interaction with the others.

The question remains: should you use virtualenv with your Graphite installation?

The obvious answer, in my opinion, is a resounding *yes*. Graphite's components have a non-trivial number of Python dependencies, many of them "locked" to a specific version number. Being able to contain these within the Graphite directory structure means you're reasonably protected from changes made to your system-wide Python toolchain. The only caveat is to remember to adjust your `PATH` to always load your Graphite `bin` directory ahead of your other paths. Don't worry, I'll remind you when it's time.

Graphite versions 0.9.13 and older are hardcoded to install to `/opt/graphite`. This will change in the next major release. Until then, however, it means although we can use Virtualenv, it *must* use `/opt/graphite` as our base installation path. If you wish to try out a development version of Graphite, you can use Virtualenv to install Graphite anywhere, including your own home directory.

Using sudo effectively

Throughout the examples below you'll notice that I run any privileged commands (where we need root permissions) using the `sudo` command. This is a widely accepted best practice, preferred over logging in as the superuser and running the commands in a full login shell. However, in order to maintain a high level of security, the default sudo configuration often disables the ability to "pass through" environment variables (such as `PATH`) to the intended user.

Because we're using virtualenv, it's vital that we maintain our custom PATH to ensure that we're using the intended `python` and `pip` binaries. We can take a couple quick steps to enable this behavior. The first is to disable the `secure_path` option in your `/etc/sudoers` file (edit with `sudo visudo`). The next is to run any affected sudo commands with the `-E` argument which preserves the existing user environment variables.

Your business or organization may have policies that restrict these sorts of changes to your sudo configuration. Fortunately there are a variety of methods you can use to achieve the intended goal: installing Graphite in a system-wide location (`/opt/graphite`) and setting our PATH to work properly with virtualenv. You can follow the examples I've provided or you can adapt your own best practices to meet these requirements.

Dependencies

There are a handful of software dependencies we'll need to install before we get to the actual Graphite bits. Unlike the rest of our Graphite files and Python libraries, these will be installed from the operating system vendor's package repository.

Debian/Ubuntu

Installing the packages on modern Debian or Ubuntu systems is straightforward. They already provide all of the dependencies we need in the official Deb repositories.

```
$ sudo apt-get install -y libcairo2-dev \
                          libffi-dev \
                          pkg-config \
                          python-dev \
                          python-pip \
                          virtualenv \
                          fontconfig \
                          apache2 \
                          libapache2-mod-wsgi \
                          memcached \
                          librrd-dev \
                          libldap2-dev \
                          libsasl2-dev \
                          git-core \
                          gcc
```

RHEL/Fedora/CentOS

The dependencies for RPM-based systems are similar, with one exception. We'll first need to install the "Extra Packages for Enterprise Linux" (EPEL) repository metadata where the `python-pip` package is distributed.

```
$ sudo yum install -y epel-release
```

```
$ sudo yum install -y cairo-devel \
                      libffi-devel \
                      pkgconfig \
                      python-devel \
                      python-pip \
                      python-virtualenv \
                      fontconfig \
                      httpd \
                      mod_wsgi \
                      memcached \
                      rrdtool-devel \
                      openldap-devel \
                      cyrus-sasl-devel \
                      git \
                      gcc
```

Installing from Source

With the system-level dependencies complete, we can move on to the fun stuff. We're going to start by creating a virtualenv environment at `/opt/graphite` where all of our components will be installed. Once this is ready we'll clone each of the projects from their GitHub repositories to the local filesystem, select the version we want to install with `git checkout`, and then install each component with their `setup.py` script.

Suffice it to say that you need to run each installation step in the order I've described them below. However, as long as you create your virtualenv first, you can always safely `rm -rf /opt/graphite` and start over from scratch if something goes awry.

Many of the following commands will log a *bunch* of output to the screen. Under normal circumstances this content can be safely ignored. If there's a serious problem the command should die and provide some errors or warnings that can be useful in searching for answers online. For the sake of brevity I'm only listing the commands below.

First things first, we need to create our working environment where the Graphite components and their Python dependencies will reside.

```
$ sudo virtualenv /opt/graphite
$ export PATH=/opt/graphite/bin:$PATH
```

Next we'll *clone* the Whisper repository, *checkout* the desired version, and run the installation script. We install Whisper first because it also happens to be a dependency for Carbon and Graphite-Web.

```
$ cd /usr/local/src
$ sudo git clone https://github.com/graphite-project/whisper.git
```

```
$ cd whisper
$ sudo git checkout 0.9.13
$ sudo -E python setup.py install
```

Logically it makes sense to handle Carbon next, although it really doesn't matter. Note the appearance of the `pip install -r` command, which resolves all Python dependencies listed in `requirements.txt`. Everything else is the same process as before.

```
$ cd /usr/local/src
$ sudo git clone https://github.com/graphite-project/carbon.git
$ cd carbon
$ sudo git checkout 0.9.13
$ sudo -E pip install -r requirements.txt
$ sudo -E python setup.py install
```

Finally, we install Graphite-Web using the same method. A couple of new commands here to handle optional dependencies for `rrdtool` and `python-ldap`. If you don't need to read from RRD files or use LDAP authentication, you can safely skip that command. The `check-dependencies.py` script is handy for testing that our Python dependencies installed successfully. If you choose to skip the RRD and LDAP packages you'll see warnings but they should be marked as `OPTIONAL` and can be safely ignored.

```
$ cd /usr/local/src
$ sudo git clone https://github.com/graphite-project/graphite-web.git
$ cd graphite-web
$ sudo git checkout 0.9.13
$ sudo -E pip install -r requirements.txt
$ sudo -E pip install rrdtool python-ldap
$ python check-dependencies.py
$ sudo -E python setup.py install
```

At this point you're effectively done *installing* Graphite. Congratulations! We still have a little ways to go before your Graphite server is ready to use, but the "scary" part is behind us.

Of course, if you chose to run the Synthesize project, you're probably already collecting metrics and graphing charts by now. That's ok, I still heart you. Let's move on to setting up our web database.

Preparing your web database

Graphite-Web uses a relational database to store graph and dashboard configuration, user accounts (if you choose to authenticate against the database), and even *Events* data. As I mentioned in the previous chapter, SQLite is not a good choice for even moderately-busy production Graphite services, but it's fine for development and test systems.

PostgreSQL fans need not worry - we'll cover the use of PostgreSQL for High-Availability clusters in our *Scaling Graphite* chapter later in this book. But for now we're going to focus on getting up and running with SQLite.

Django uses a database schema format known as *fixtures*. Graphite-Web includes one of these fixtures that holds all of the data and relationships it needs to support the aforementioned feature set.

Running the `manage.py syncdb` command below will install the fixtures and also prompt you to create a superuser account. You'll want to do this now so you can administer your web database later. You should also see a warning about setting your `SECRET_KEY`. This is an important security step that we'll address later when we configure Graphite-Web's `local_settings.py` file.

```
$ cd /opt/graphite/webapp/graphite

$ sudo -E python manage.py syncdb

Could not import graphite.local_settings, using defaults!
/opt/graphite/webapp/graphite/settings.py:244: UserWarning: SECRET_KEY is set to an unsafe default
  warn('SECRET_KEY is set to an unsafe default. This should be set in local_settings.py for better
Creating tables ...
...
... snipped for brevity ...
...
Creating table tagging_taggeditem

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'root'): admin
E-mail address: admin@mycompany.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

At this point your web database is ready to use. Like any database, it's a good idea to keep backups for disaster recovery. This process is out of the scope of this book but it's no different than backing up any other common database schema.

Configuring Carbon

Although Carbon uses up to eight different configuration files, only two of them are mandatory: `carbon.conf` and `storage-schemas.conf`. The former manages how many of each different Carbon daemons to run, which addresses and ports they listen on, and many of the settings that define their behavior, influence performance, and enable debugging. The latter defines the retention policy (precision and duration) for all Whisper metrics.

We're going to look at an example of each that would be suitable for running on a small production Graphite server. You're welcome to transcribe these settings verbatim, or you can copy the `*.example` versions that Carbon includes at installation, and make the changes I've recommended below.

Note that changes to the following files will only take effect after restarting the affected Carbon daemons.

carbon.conf

Most of Carbon's configuration files use the INI file format, and `carbon.conf` is no exception. Each Carbon daemon type will have its own section starting with a bracketed header. If you're running multiple instance of a certain type, each process will have its own section with the header specifying an alphanumeric identifier or index (e.g. `[cache:1]`).

Section Inheritance

The `carbon.conf` file supports an INI feature known as *section inheritance*. Each per-instance section inherits settings from the main daemon section preceding it. In our sample configuration, `[cache:1]` will inherit all of the settings specified in the `[cache]` section.

A single-cache configuration like this doesn't really take advantage of this feature. But as you begin to scale up the number of processes, you'll appreciate the way section inheritance eliminates redundant settings in your configuration. For example, it would only take four lines to add another cache instance to this configuration.

Example 4-2. carbon.conf

```
[cache]
USER = carbon
MAX_CACHE_SIZE = inf
MAX_CREATES_PER_MINUTE = 100
MAX_UPDATES_PER_SECOND = 1000
LINE_RECEIVER_INTERFACE = 0.0.0.0
```

```
PICKLE_RECEIVER_INTERFACE = 0.0.0.0
CACHE_QUERY_INTERFACE = 0.0.0.0
LOG_CACHE_HITS = False
LOG_CACHE_QUEUE_SORTS = False
LOG_UPDATES = False

[cache:1]
LINE_RECEIVER_PORT = 2003
PICKLE_RECEIVER_PORT = 2004
CACHE_QUERY_PORT = 7002
```

This configuration is much simpler than you're bound to use in production, but it also includes everything that should be important to help us get started. Graphite developers aim to use sane defaults where possible and avoid breakage (e.g. during upgrades) at all costs. In a similar vein, I've avoided listing a lot of the boilerplate where their meanings and usage should be obvious; rather, I'd like to highlight a few key settings that should be relevant to any new user.

`USER`
: When this value is defined the specified Carbon daemon will run as this user. Setting this to an unprivileged user is a good practice to adopt. However, it also means that we'll need to create the specified `carbon` user and group, as well as adjusting some directory permissions (we'll get around to this in the next section). Note that `USER` is unset by default, which makes me sad, but is a *reasonable* default in helping users to get started when they don't have a Graphite book at their disposal (wink).

`MAX_CACHE_SIZE`
: This setting determines the amount of memory that `carbon-cache` should be allowed to use for caching. The default value (*inf*) assumes that your disk subsystem can keep up with the amount of *creates* and *updates* performed by `carbon-cache`. If your write performance is poor, it's a good idea to set this to an finite number representing the *number of datapoints* to store in memory (**not** the size in bytes). If the cache hits this limit, datapoints may be lost until the buffer catches up on writes. Each datapoint in Whisper is 12 bytes large, so you should be able to calculate some safe numbers based on your available memory. If you're running Graphite in a heavily utilized virtual host, I strongly encourage you to set `MAX_CACHE_SIZE` to a non-*inf* value.

`MAX_CREATES_PER_MINUTE`
: Unsurprisingly, this limits the number of new Whisper files that `carbon-cache` will create *per minute*. If this limit is reached in any one-minute window, any datapoints that arrive for a new metric will be dropped on floor. That metric will be created the next time a datapoint arrives for that metric *and* this limit has not been reached. It can be tempting to set this value artificially high to guarantee new metrics, but you'll end up shooting yourself in the foot since *creates* are significantly more expensive to process than *updates*.

`MAX_UPDATES_PER_SECOND`
: This setting limits the number of Whisper updates performed by `carbon-cache` ***per second***. Although it might seem confusing to measure creates per minute and updates per second, it makes sense when you consider that updates come in *much more frequently* than creates. When `carbon-cache` hits this limit, it responds by batching up multiple writes at once for efficiency.

`LINE_RECEIVER_INTERFACE`
: The IPv4 or IPv6 network address that Carbon should listen on for *plaintext* metrics. Valid for all Carbon daemon types. Defaults to `0.0.0.0`.

`PICKLE_RECEIVER_INTERFACE`
: The IPv4 or IPv6 network address that Carbon should listen on for *pickled* metrics. Valid for all Carbon daemon types. Defaults to `0.0.0.0`.

`CACHE_QUERY_INTERFACE`
: The IPv4 or IPv6 network address that the `carbon-cache` daemon should listen on for cache queries (i.e. from the web application). Defaults to `0.0.0.0`.

`LOG_CACHE_HITS`, `LOG_CACHE_QUEUE_SORTS`, *and* `LOG_UPDATES`
: These control various aspects of logging for `carbon-cache` and tend to be overly noisy. They can theoretically have an impact on the write volume of a Carbon host (particularly a very busy one) so I recommend disabling them to start.

Default Carbon settings

If you're ever unsure about a Carbon configuration setting, check the defaults in `lib/carbon/conf.py`. Using the source installation method we performed in this chapter, you'll find this file at `/opt/graphite/lib/carbon/conf.py`. Open it in your favorite editor or pager utility (e.g. `more` or `less`) and look for the `defaults` object (in Pythonese, it's referred to as a *dict*).

storage-schemas.conf

This file controls the retention policies for all of your metrics. It uses the same INI format as `carbon.conf`, although the section header names are arbitrary and used only to describe each schema (although they must be unique). Every schema section needs to contain a *pattern* and one or more comma-delimited *retentions*.

Each pattern defines a regular expression to match on some portion (or the entire string) of a metric name. Carbon uses Python's built-in regular expression syntax, which should be similar to anyone who's used Perl-style expression syntax.

Retentions describe the precision and length of time to store each datapoint. Values can be represented as a raw integer or *time description* using the following units: *seconds*,

minutes, *hours*, *days*, *weeks*, and *years*. The *months* descriptor is not supported since they contain an irregular number of days. The following expressions are all equivalent:

```
10:604800
10s:7d
10seconds:1week
```

I've defined two different schemas in the following `storage-schemas.conf`. The first one matches any metrics with the `collectd.` prefix (you might not have *collectd* running on your Graphite server, but that's ok; it won't hurt anything to include it). These metrics will be stored at two retention levels: ten-second precision for one week, and one-minute precision for one year. The former is considered the *raw precision* since we're storing the original values intact. The latter is being downsampled into a *rollup*; as new metrics arrive, the rollup archive will continually be recalculated according to that metric's *aggregation method*.

Example 4-3. storage-schemas.conf

```
[collectd]
pattern = ^collectd\.
retentions = 10s:1w, 60s:1y

[default]
pattern = .*
retentions = 60s:1y
```

The second schema acts as a default schema for any metrics that didn't already match a pattern defined before this one. Remember, the name "default" is completely arbitrary; it's the *pattern* that determines it will match all metrics. The net result of our schema policy here is that all metrics will be stored at one-minute precision for one year, except for *collectd*-originating metrics, which will also be stored at ten-second precision for a week.

Settings in your `storage-schemas.conf` only affect Whisper files *during creation*. If you decide to change a retention policy after the corresponding metric file has already been created, you'll either need to remove the affected Whisper file and let it recreate with the new policy or use the `whisper-resize.py` command to change the retention(s) manually.

storage-aggregation.conf

I hadn't intended to cover this particular configuration file since it's considered optional, but it really does play an important part if you're intending to send any counter metrics or pre-aggregated metrics (e.g. using *StatsD*) to your Carbon service. You can consider

this section a bonus, but please don't tell my editor since I've asked to be paid by the word.

In the preceding section I mentioned that the *aggregation method* plays an effect in how we *rollup* metrics into lower-precision archives. Well, this is the file that allows you to specify the aggregation policy for your metrics. The syntax is very similar to `storage-schemas.conf`, with the addition of the *xFilesFactor* and *aggregationMethod* directives. We already covered the purpose of each in the preceding chapter, so I urge you to jump back there if you need a quick refresher.

If you're not already using an aggregator like *StatsD* that emits metrics like *min*, *max* and *count*, you probably won't need this file just to get started, but I think it merits a mention in this section anyways.

The same rules apply to `storage-aggregation.conf` regarding changes to an existing Whisper file; if you need to change the *xFilesFactor* or *aggregationMethod*, you'll need to use `whisper-resize.py` to reconfigure the archives manually.

Example 4-4. storage-aggregation.conf

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

[max]
pattern = \.max$
xFilesFactor = 0.1
aggregationMethod = max

[sum]
pattern = \.count$
xFilesFactor = 0
aggregationMethod = sum

[default_average]
pattern = .*
xFilesFactor = 0.5
aggregationMethod = average
```

Some final preparations

If you followed my suggestion to run your Carbon daemons as an unprivileged `USER`, I applaud you. However, there are a few final housekeeping tasks we'll need to perform before Carbon is ready to start (if you decided to run Carbon as a superuser, shame on you but you can skip this section).

First, we'll need to create the system user.

```
$ sudo /sbin/groupadd carbon
$ sudo /sbin/useradd -c "Carbon user" -g carbon -s /bin/false carbon
```

We also need to fix the ownership permissions for a number of storage and log directories. These allow the various Carbon daemons to write logs as the unprivileged user, create and update Whisper files, and in one case, guarantees that the webserver user (in this case, `www-data`; adjust as necessary based on your distribution) will be able to write to the SQLite web database.

```
$ sudo chmod 775 /opt/graphite/storage
$ sudo chown www-data:carbon /opt/graphite/storage
$ sudo chown -R carbon /opt/graphite/storage/whisper
$ sudo mkdir -p /opt/graphite/storage/log/carbon-{cache,relay,aggregator}
$ sudo chown -R carbon:carbon /opt/graphite/storage/log
```

Starting your Carbon daemons

You should now have at minimum, a `carbon.conf` and `storage-schemas.conf` saved in your configuration directory (`/opt/graphite/storage/conf`, if you followed my instructions). You've also created a dedicated user to run your Carbon daemons under, and prepped your storage and log directories accordingly. We're finally ready to start your Carbon service.

If you configured the single `carbon-cache` instance like we covered earlier, starting the process is as simple as:

```
$ sudo -E carbon-cache.py --instance=1 start
```

Many online examples use letters to distinguish between instances (a, b, c, etc.). However, I recommend using numbers to identify your daemon instances instead. This pattern is much easier to script or automate when you need to manage a sequence of processes. Say we've scaled up our number of cache processes to handle an expected spike in traffic. Starting them all at once would be as straightforward as:

```
$ for i in `seq 8`; do sudo -E carbon-cache.py --instance=${i} start; done
Starting carbon-cache (instance 1)
Starting carbon-cache (instance 2)
Starting carbon-cache (instance 3)
Starting carbon-cache (instance 4)
Starting carbon-cache (instance 5)
Starting carbon-cache (instance 6)
Starting carbon-cache (instance 7)
Starting carbon-cache (instance 8)
```

Stopping them is the same command, except using the `stop` action.

```
$ for i in `seq 8`; do sudo -E carbon-cache.py --instance=${i} stop; done
Sending kill signal to pid 20961
Sending kill signal to pid 20968
Sending kill signal to pid 20973
Sending kill signal to pid 20977
Sending kill signal to pid 20981
Sending kill signal to pid 20987
Sending kill signal to pid 20992
Sending kill signal to pid 20998
```

Although these examples only detail how to start and stop the `carbon-cache` daemon, these instructions are exactly the same for the `carbon-relay` and `carbon-aggregator` daemons.

Configuring Graphite-Web

Setting up your Graphite web application is pretty much a walk in the park after dealing with the myriad configuration files and settings for the various Carbon daemons. There are only a few files we'll need to prepare. If you're still following along at home, we should be up and running in a few minutes.

local_settings.py

All of the Graphite's web application settings are configured in the `local_settings.py` file, which in turn overrides the defaults in the `settings.py` file. **Do not** make changes to `settings.py` yourself, as this file gets overwritten during upgrades. We'll get started by copying the sample file provided by Graphite-Web and then making our changes there.

```
$ cd /opt/graphite/webapp/graphite
$ sudo cp local_settings.py.example local_settings.py
```

The following settings are really the only ones that we *need* to set before we start using Graphite-Web. Truth be told, `SECRET_KEY` is the only **mandatory** edit, although there's a very good chance that you'll want to set `TIME_ZONE` to avoid any possible confusion in your graphs. And enabling memcached here with `MEMCACHE_HOSTS` is such an easy performance win (when you finally need it) that it'd be silly *not* to do it now.

`SECRET_KEY`
: This value is used for salting hashes used in various aspects of the application security. It should be set to a reasonably long and random value, such as the string output of `date | sha256sum`. Note that if you happen to be running multiple web

application servers behind a load-balancer, this setting should be identical across servers.

`TIME_ZONE`
: Controls the time zone for metric queries or rendered graphs. Because all data handling is performed server-side, this will affect virtually all aspects of the user experience. If you have users across multiple time zones, it's probably best to leave this at the default value (`UTC`).

`MEMCACHE_HOSTS`
: A list of memcached servers to be used for caching metric query results. For our purposes we need to set this to `['127.0.0.1:11211']`, where our memcached service is running by default.

After editing those values in `local_settings.py`, we just have one last task before we proceed with setting up the Apache configuration. The default `LOG_DIR` location is fine, but the permissions and ownership wouldn't allow the Apache system user to write the necessary log files in place. We'll just need to change ownership on that directory before we continue.

```
$ sudo chown www-data /opt/graphite/storage/log/webapp  # Debian and Ubuntu
$ sudo chown apache /opt/graphite/storage/log/webapp    # RHEL, Fedora and CentOS
```

Setting up Apache

As you may recall, we're using Apache's mod_wsgi to run Graphite as a WSGI application. We need a simple Python script dropped into place to let Apache know where to find all the relevant Graphite and Django files to load at runtime. Fortunately, the Graphite-Web project already includes a working example that we can copy without any changes.

```
$ cd /opt/graphite/conf
$ sudo cp graphite.wsgi.example graphite.wsgi
```

Next we need to create a *VirtualHost* configuration for Apache. This example is intentionally simple, providing just the bare necessities to allow us to get Graphite running under the Apache webserver. Note that the access control directives below may vary depending on the version of Apache your distribution offers.

This file should be saved or copied into place wherever your distribution prefers to store its Apache configuration files. For Debian or Ubuntu, this typically means saving it as `/etc/apache2/sites-available/graphite.conf` and enabling the configuration with `a2ensite graphite`. For RHEL, Fedora or CentOS users, you probably want to save it as `/etc/httpd/conf.d/graphite.conf`.

```
<VirtualHost *:80>

  # wsgi script and permissions to read it
  WSGIScriptAlias / /opt/graphite/conf/graphite.wsgi
  <Directory /opt/graphite/conf>
    Require all granted
  </Directory>

</VirtualHost>
```

There are quite a few other best practices I'd normally apply here but we've kept it basic for the sake of expediency. Most administrators would be wise to enable SSL and some sort of basic authentication to keep out prying eyes.

Now we're ready to start the webserver.

```
$ sudo service apache2 start  # Debian and Ubuntu
$ sudo service httpd start    # RHEL, Fedora and CentOS
```

You may prefer to run Graphite as a WSGI application under NGINX instead of Apache. Although we're not going to cover that particular setup in this book, rest assured there are a number of good resources and examples online to help you get running in no time.

Verifying your Graphite installation

> You cannot tell, but I am giving you a thumbs up.
>
> — N.E.P.T.R.

At this point you should have a working Graphite installation with Carbon and Graphite-Web. Whether you got here the *easy way* using Synthesize and Vagrant, or taking the *deliberate route* with a hands-on manual installation, I applaud you for taking the initiative to set up your own personal metrics collection and visualization engine. It's about to get crazy fun all up in here, at least if you care about data and graphs like I do.

Carbon Statistics

Even though you're probably not sending metrics to the Carbon listener yet, each of the Carbon daemons records its own individual metrics to Whisper. These are an easy way to verify that `carbon-cache` is *doing something*. But perhaps we should first just confirm that `carbon-cache` is running at all.

```
$ cat /opt/graphite/storage/carbon-cache-1.pid
11730

$ pgrep -fl carbon
11730 carbon-cache.py
```

Ok, that looks perfect. We should probably also do a quick spot-check of the network ports that `carbon-cache` *should* be listening on.

```
$ netstat -vant | grep LISTEN | grep '[27]00'
tcp        0      0 0.0.0.0:7002            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:2003            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:2004            0.0.0.0:*               LISTEN
```

Good, we can see that `carbon-cache` has the *plaintext* listener on port **2003**, the *pickle* listener on port **2004**, and the *cache query* port on **7002**. Now let's check out the Whisper filesystem and make sure that Carbon is writing its own statistics out to disk.

```
$ cd /opt/graphite/storage/whisper

$ ls -l
total 4
drwxr-xr-x 3 carbon carbon 4096 Jan  2 00:16 carbon

$ find carbon
carbon
carbon/agents
carbon/agents/synthesize-1
carbon/agents/synthesize-1/creates.wsp
carbon/agents/synthesize-1/memUsage.wsp
carbon/agents/synthesize-1/blacklistMatches.wsp
carbon/agents/synthesize-1/whitelistRejects.wsp
carbon/agents/synthesize-1/cache
carbon/agents/synthesize-1/cache/bulk_queries.wsp
carbon/agents/synthesize-1/cache/size.wsp
carbon/agents/synthesize-1/cache/queues.wsp
carbon/agents/synthesize-1/cache/queries.wsp
carbon/agents/synthesize-1/cache/overflow.wsp
carbon/agents/synthesize-1/cpuUsage.wsp
carbon/agents/synthesize-1/updateOperations.wsp
carbon/agents/synthesize-1/metricsReceived.wsp
carbon/agents/synthesize-1/committedPoints.wsp
carbon/agents/synthesize-1/avgUpdateTime.wsp
carbon/agents/synthesize-1/errors.wsp
carbon/agents/synthesize-1/pointsPerUpdate.wsp
```

Sure enough, `carbon-cache` is writing out all of its internal stats. Heck, we can even look at the last five datapoints for the *cpuUsage* on this `carbon-cache`:

```
$ whisper-fetch.py carbon/agents/synthesize-1/cpuUsage.wsp | tail -5
1422938640  0.795059
1422938700  0.793212
1422938760  0.843837
1422938820  0.799686
1422938880  0.783158
```

Feeding new data to Carbon

That's all Kool and the Gang, but what about creating our own bespoke metrics? Let's fire off a simple test datapoint off to the `carbon-cache` plaintext port and see what happens. If everything's working as intended, we should have a new Whisper file with the datapoint recorded and available for retrieval.

```
$ echo "test.foo 1 `date +%s`" | nc localhost 2003

$ pwd
/opt/graphite/storage/whisper

$ ls -l
total 8
drwxr-xr-x 3 carbon carbon 4096 Jan  2 00:16 carbon
drwxr-xr-x 2 carbon carbon 4096 Feb  3 00:11 test

$ ls -l test/
total 6160
-rw-r--r-- 1 carbon carbon 6307228 Feb  3 00:11 foo.wsp

$ whisper-fetch.py test/foo.wsp | tail -5
1422940020  None
1422940080  None
1422940140  None
1422940200  None
1422940260  1.000000
```

Building your first graph

Are you anxious to start graphing some of these metrics yet? I'm really excited you've made it this far because this is where Graphite *really* turns the fun up to `11.0`. Now that we've got some metrics in the system let's see what they look like all tarted up as line charts.

For what it's worth, the graphs you're about to see came from one of my Synthesize hosts, so you're bound to see less (and different) metric history than mine. That's ok, you know what they say: *it's not the size of your metrics history, it's how you use it to support your incident response*.

We're going to walk through a few steps just to verify that Graphite-Web is working as intended. If you're new to Graphite this will also give you your first hands-on experience with the user interface.

Loading the web interface

Synthesize users who are using the Vagrant method should be able to load the Graphite window in your favorite browser at *https://127.0.0.1:8443/*. Other Synthesize users will

need to connect to the HTTPS service at the IP address (or hostname) of the system where you ran the installation script. If you followed the instructions in this chapter, you should be able to load Graphite on the HTTP port of the intended system.

The first time you load the Graphite user interface you'll notice that it uses a somewhat conventional application layout: a navigation tree in the side panel to the left, a main panel containing the *Composer* window, and a title header at the top containing a series of links (e.g. *Login*, *Documentation*, *Dashboard*, etc).

Most users should be familiar with the navigational icons and elements found here. Within the navigation tree you'll encounter nested folders such as *Metrics* that represent the structure of Whisper directories and files stored on the local filesystem. Not surprisingly, these also map to the hierarchy of dot-delimited metric names submitted to our Graphite server (or in the example below, emitted by `carbon-cache`).

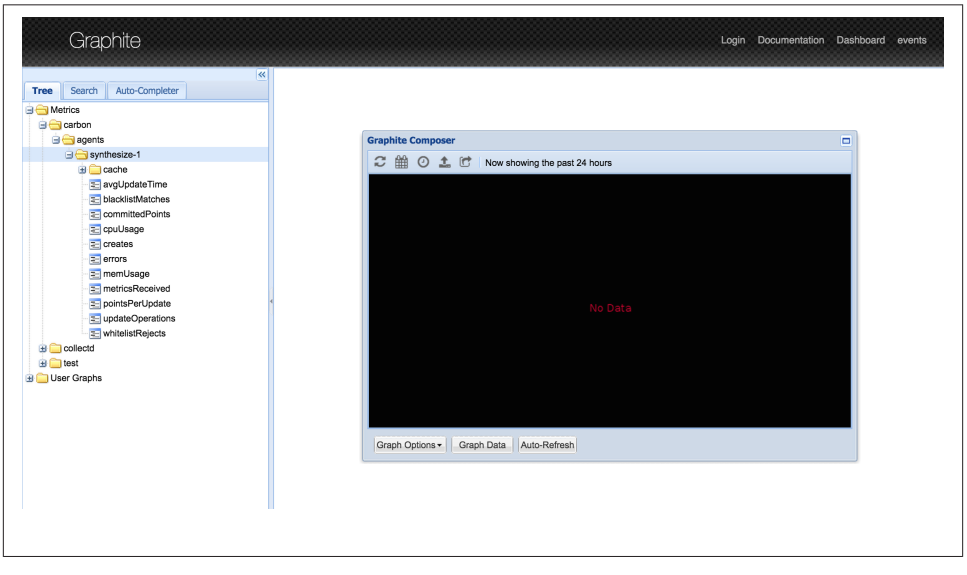


Figure 4-1. Navigating metrics in the Tree

The first time you click on any of these folders (until the next page reload), the Graphite interface (i.e. *client*) makes an *AJAX* call to the backend Graphite web service, asking for any known objects (i.e. child folders or leaf elements, such as metrics). The service then returns a *JSON* object containing what it knows, which the client interprets and uses to redraw the tree view. If you're familiar with the *web inspector* or *developer view* of modern browsers, you can even inspect the page yourself to watch the requests and responses; this is a great way to learn how Graphite-Web interacts with the underlying components.

Why do I mention all this gobbledygook? Because if this simple navigational example works, it's your first hint that Graphite-Web, Carbon, and Whisper are all working in tandem. It means that we have Whisper metrics stored on disk and that the webserver's system user has the necessary permissions to read them. *High Five*.

Adding Metrics to a Graph

Clicking on any metric node in the *Metrics* tree will add it to graph embedded in the *Composer* window in the main panel. If the metric already exists in the graph (and hasn't been modified with render functions), clicking on the metric node will remove it from the graph.

The Composer will try to keep you from adding the same metric unnecessarily to a graph. If you really need to render the same metric multiple times in a single graph (e.g. to compare to a timeshifted period), apply at least one render function to the first metric instance. This makes the metric definition unique, allowing you to add the same metric a second time by clicking on the node in the Metrics tree.

Here I've selected the `metricsReceived` metric listed under the `carbon.agents.synthesize-1` namespace. A line series representing the available datapoints is rendered in the graph, and a simple legend appears at the bottom of the graph associating the metric with the color blue.

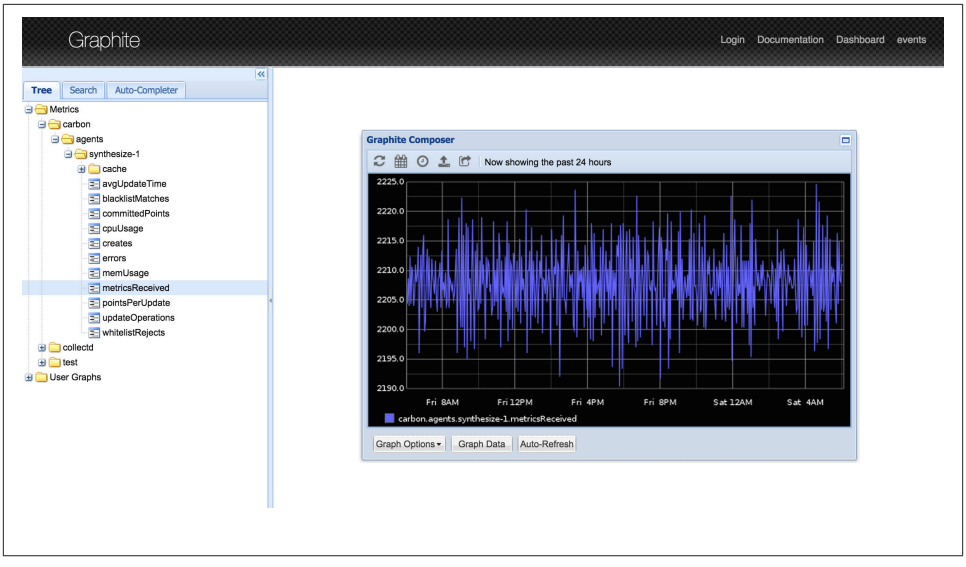


Figure 4-2. Adding a metric to your graph

By also selecting the `updateOperations` metric, we can see in Figure 4-3 that our graph now contains two line series. The new metric has been associated with the color green, and both metrics are included in the legend.

You may have also noticed that the scale of our Y-axis within the graph has changed substantially. Graphite is smart enough to automatically scale the axis so that every metric is visually represented in the image. If it wasn't, that large *dip* in `updateOperations` would've been cut off by the lower boundary of our graph.

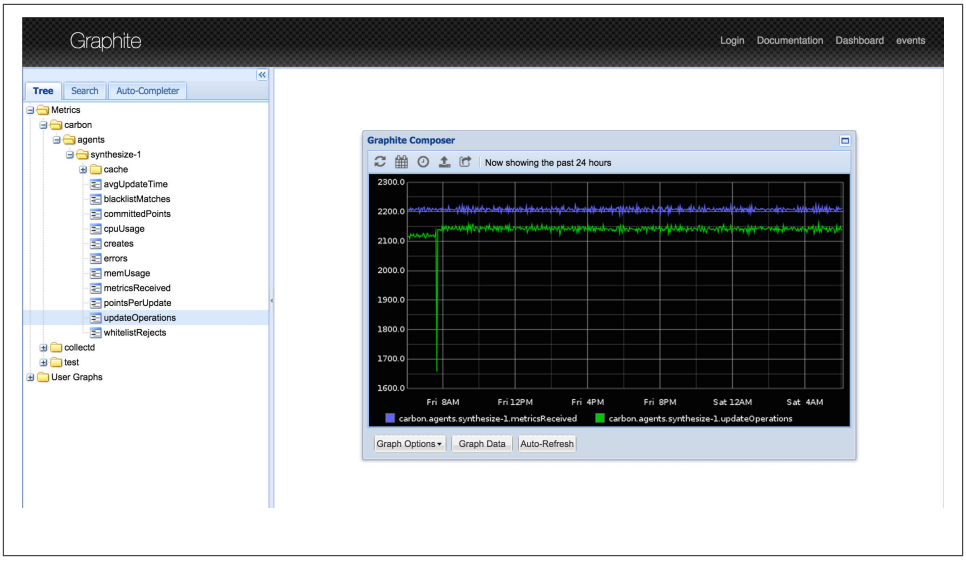


Figure 4-3. Adding another metric

Now is a great time to wind things up and regard our installation as a success. There's a lot of exciting stuff ahead of us in the next chapter, so we'll resume our graph party over there.

I truly hope you picked up a few useful tips in this chapter. We covered a lot of ground in terms of configuration settings and system commands, although this process was simplified a great deal to make it palatable to new users. We'll cover far more advanced settings and configurations later on in this book, but it's important that we're able to lay a basic foundation and move forward with the interesting content for now.

CHAPTER 5
The Graphite User Interface

We're living in an age of beautiful application design. It took years to get to this point, but online designers finally understand how to take full advantage of web browsers and modern web technologies to create innovative and uniquely creative interfaces, blending the best of interactive art and utility. There are a myriad of examples that we can look to for inspiration in our daily lives.

Graphite is not one of them.

Don't get me wrong, I love Graphite. It is a fantastically flexible and functional tool for building graphs and dashboards. It's ugly, but in a way that belies the beauty of its API underpinning. It exposes a wealth of functionality in a sane and intuitive manner. Virtually every feature found in the rendering API is available within the Graphite user interface without becoming cumbersome or confusing; considering the extent of this feature set, this is a laudable accomplishment on its own.

That the Graphite UI resembles something out of Redmond, Washington circa 2000 is merely an aesthetic speed bump on our journey to graphing mastery. And while most of your dashboards will be run in a third-party application or built by embedding graphs in your own web pages, there's a very good chance that for ad-hoc graph prototyping or metrics introspection, you'll be hard-pressed to beat the simplicity of the original Graphite interface.

Much of this chapter is devoted to learning how to navigate the Graphite UI and getting up to speed on the vast array of chart-level rendering options available. Although each section is dedicated to learning a different aspect of the interface, we'll be applying many of these lessons in the context of graphing Carbon's own internal statistics. By the end of the chapter you should be able to construct a variety of line and area charts from scratch, save a graph, retrieve a saved graph, and share a graph URL with a friend or loved one.

Finding Metrics

Graphite offers a few different techniques for listing or searching for metrics in the user interface. Each method is presented as one of the tabbed views in the left-hand navigation pane. They're plainly labeled according to their respective functionality: *Tree*, *Search*, and *Auto-Completer*.

Navigating the Tree

The navigation Tree is probably one of the first page elements you notice upon loading the Graphite UI. If you haven't logged into a Graphite user account (there's a good chance you haven't yet), you'll see two folders: *Metrics* and *User Graphs*. The former contains a hierarchical tree of all available metrics available to Graphite. Clicking on a folder (or *node*) will expand the folder, revealing either more folders, some metrics, or a combination of both.

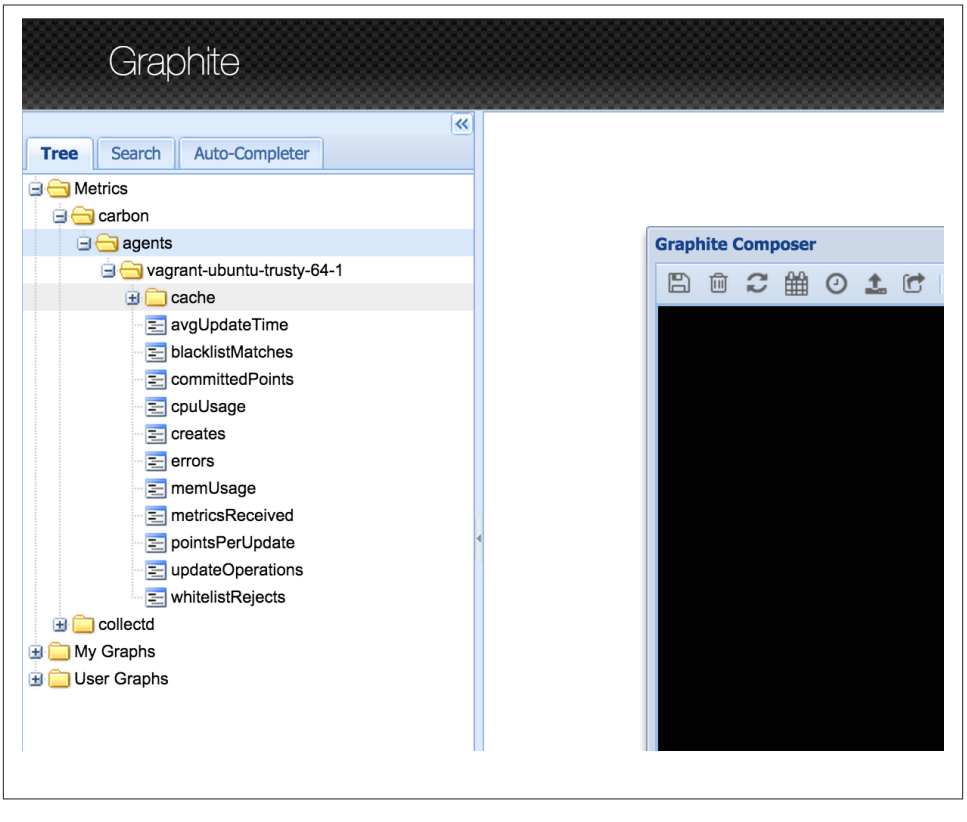


Figure 5-1. Navigating the Metrics branch

Metrics

As you might remember from our earlier chapters, Graphite uses a dot-delimited notation for the namespace of all metrics. This namespace corresponds to the directory and file structures in the Whisper filesystem, as well as the visual hierarchy in the *Metrics* branch of the navigation tree. Take a few minutes to explore the list of metrics and familiarize yourself with their organization.

If you skipped the previous chapter, now would be a great time to go back and quickly review the ***Building your first graph*** section near the end of Chapter Four. We did a quick walkthrough on adding metrics to create your first graph, but more importantly, it includes a few nice tips that may prove useful for troubleshooting later on.

One of the first things that's immediately obvious to new Graphite users is just how easy it is to start creating graphs from scratch. Browse through the tree, click on a metric name, and **boom**, you have a graph. Click on another metric and suddenly you have a multi-series line graph. And as we'll see in a bit, it's just as easy transforming each of those metric series to discover new insights into your data. For anyone coming from legacy monitoring or trending systems, it's a real eye-opening experience.

If you haven't tried it already, let's go ahead and create our first line graph. Open up the *Metrics* folder and drill down through the *carbon* branch until you see the list of internal `carbon-cache` metrics like we see in Figure 5-1. Click on the `metricsReceived` node and a line series, also known as a *target* in the render API, appear in the *Graphite Composer* window in the center of the main panel. It should look something like the graph in Figure 5-2.

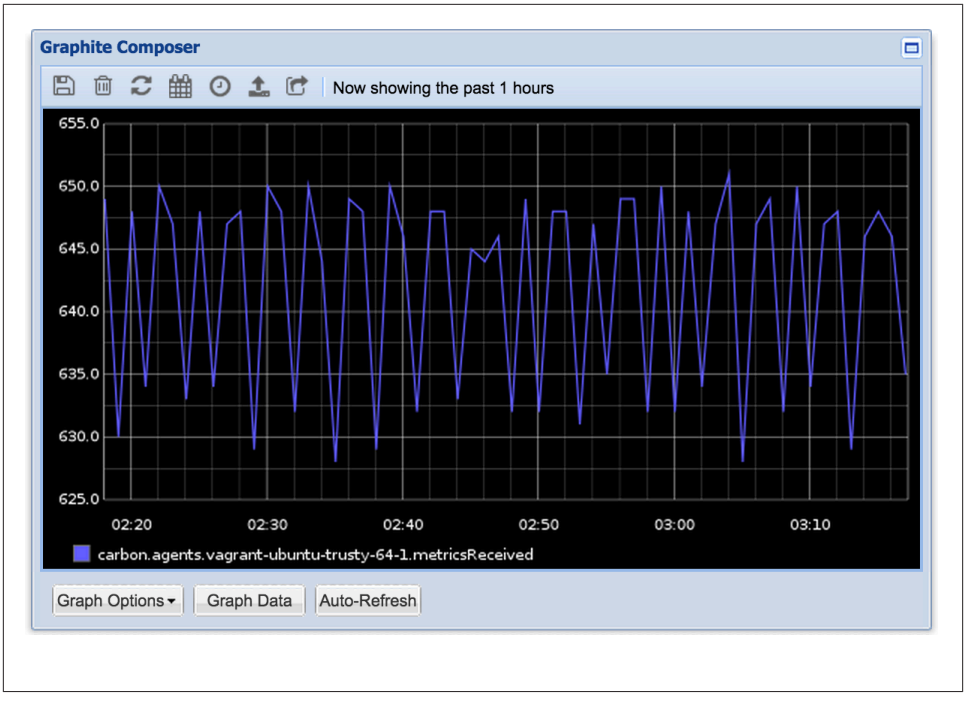


Figure 5-2. A basic line chart

MyGraphs and User Graphs

The next two folders at the root level of the navigation tree are *MyGraphs* and *User Graphs*. These are used for organizing saved graphs that you've constructed in the Composer. We'll cover saving and retrieving graphs in a bit - for now it's enough to acknowledge that this feature exists and these branches are where to look for persisted graphs when you need them.

If you're scratching your head right now because you don't see *MyGraphs*, don't worry. That's because you're not logged in with a Graphite user. Remember that superuser account we created during the installation process? If you click the *Login* link in the banner at the top and use those credentials to authenticate, you'll see the *MyGraphs* folder when you're redirected back to this page.

Anytime you reload the Graphite user interface you'll lose your existing graph. The Graphite web application does not use sessions to maintain your state or graph composition. If you're worried about losing a graph in progress, you'll want to save the graph to the Graphite database or bookmark the url of the embedded graph image (we'll cover both of these steps a little later).

As you might expect, *MyGraphs* contains all of the graphs associated with your user account (or superuser account, in this case). On the other hand, *User Graphs* contains **everyone's** saved graphs, organized by user accounts. Graphite has no access controls on these graphs, so you can browse through anyone's saved graphs, load them up, and even save them to your personal namespace.

Using the Search feature

As Graphite becomes increasingly popular within your organization, it's likely that the number of distinct metrics will explode beyond your expectations. Before long you might discover that finding specific metrics is like the metaphorical needle in a haystack. Fortunately, the *Search* tab is here to save your bacon.

Click on the *Search* tab in the navigation pane to the left and you'll find a basic input field. Enter one or more query strings, hit *enter*, and a list of matching metrics should appear in the space below the search field. The query engine is rather simple and only supports *inclusive* (or) results for multiple strings; it doesn't support any complex query languages or *union* (and) queries.

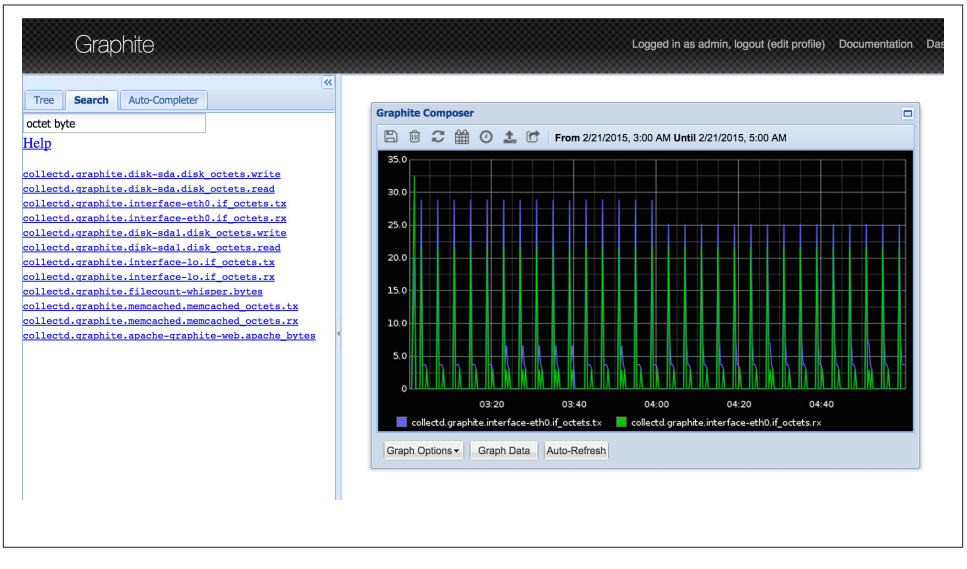


Figure 5-3. Using the Search tab to find metrics

Metrics can be added or removed to the Composer graph by clicking on any of the results, just like you might in the Tree view. But unlike the tree, the Search feature is very effective for finding those unusual or hard-to-find metrics.

It's important to note that Graphite uses a simple *index* file for this feature and that you will need to automate the regeneration of this index unless you're running Graphite

from a distribution package that handles this for you (some of them actually do). This is easy enough to verify by checking the timestamp on the index, which should exist in your `STORAGE_DIR`. If the file is *not* being updated you'll want to add a cron entry or other automated task to run the script which regenerates the index.

Example 5-1. Rebuilding the search index

```
$ ls -l /opt/graphite/storage/index
-rw-r--r-- 1 root root 5070 Feb 19 17:24 /opt/graphite/storage/index

$ sudo -E /opt/graphite/bin/build-index.sh
[Fri Feb 20 04:50:39 UTC 2015]  building index...
[Fri Feb 20 04:50:39 UTC 2015]  complete, switching to new index file

$ head -5 /opt/graphite/storage/index
carbon.agents.vagrant-ubuntu-trusty-64-1.updateOperations
carbon.agents.vagrant-ubuntu-trusty-64-1.errors
carbon.agents.vagrant-ubuntu-trusty-64-1.committedPoints
carbon.agents.vagrant-ubuntu-trusty-64-1.pointsPerUpdate
carbon.agents.vagrant-ubuntu-trusty-64-1.avgUpdateTime
```

Working smarter with the Auto-Completer

Graphite also includes an *Auto-Completer* feature that's designed for the rapid selection of metrics using as few keystrokes as possible. I warn you though - once you get used to it, you may have a hard time going back to the navigation Tree or Search field.

As you begin typing characters into the input field, Graphite performs asynchronous calls to the backend API to find any metric nodes that contain a matching prefix. Results are rendered in real-time as a drop-down list below the field. You can arrow up or down to find the next node you're interested, then hit *tab* or *enter* to populate the field with the result. When you've finally narrowed the results down to the metric you're interested in, hit *enter* to add it to or remove it from your graph.

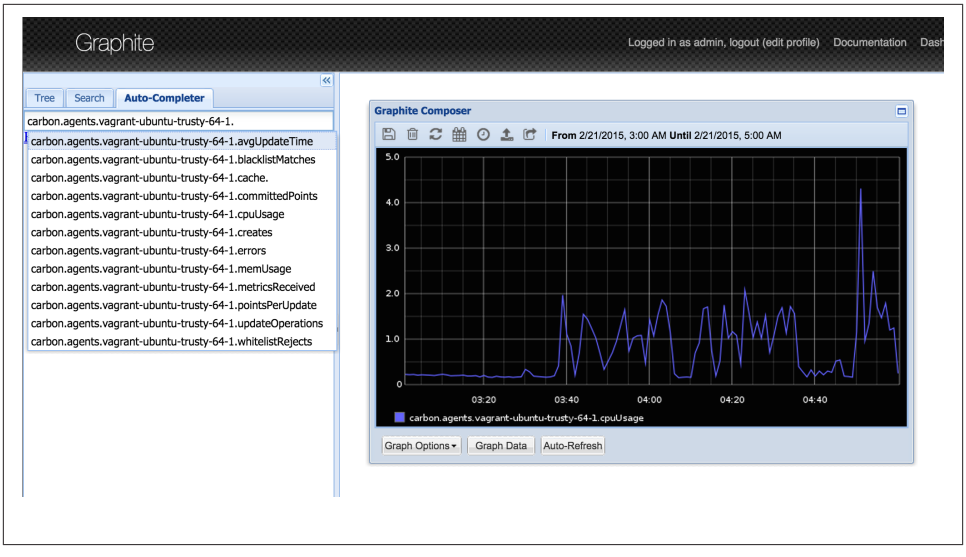


Figure 5-4. Using the auto-completer field

Wildcards

Wildcards are arguably one of Graphite's most powerful features. They make it possible to pull in collections of metrics with a single *target* definition, and combined with functions like `groupByNode()`, offer advanced grouping and organization capabilities. The most common wildcard is undoubtedly the asterisk (*) although you can also use other POSIX-compatible wildcard syntax such as lists inside curly braces (e.g. `{foo,bar}`).

The auto-completer field is one of a few places in Graphite where you can use wildcards to compose the metric *target*. For example, instead of having to manually add each of the disk metrics under the `collectd.graphite.disk-sda1` namespace, we could just define our target as `collectd.graphite.disk-sda1.*.*`. Note that Graphite treats the wildcard as "non-greedy": in our previous example it's not enough to use an asterisk to represent "the rest of the string"; you must account for the depth of dot-delimited nodes in the string definition.

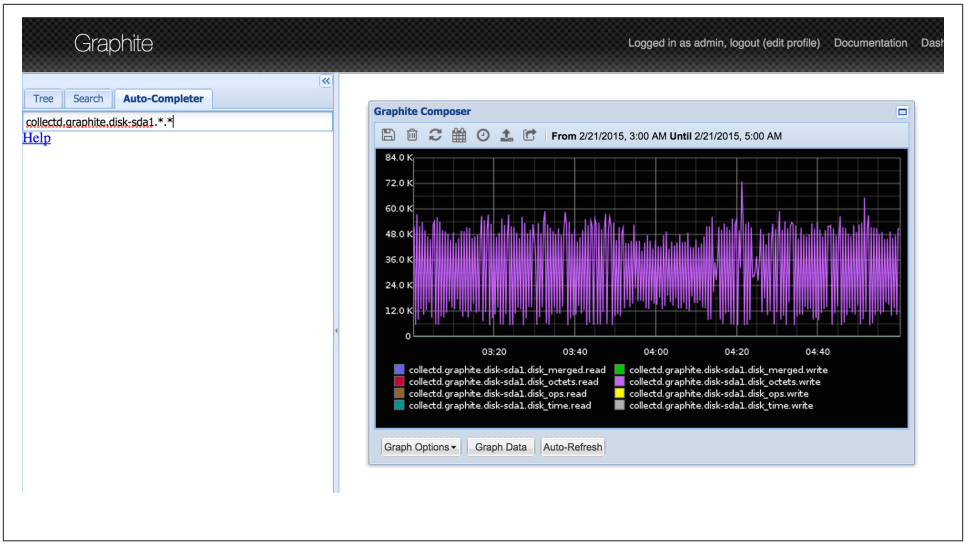


Figure 5-5. Auto-completing with wildcards

There is one little trick to remember when using wildcards in the auto-completer. When it returns the list of results matching your wildcard string, you must hit the *escape* key to remove focus from the drop-down. Otherwise you'll hit *enter* and it will add whichever metric was currently selected in the list to the graph. By escaping those selections it will then allow you to use the *enter* key to add the wildcard string as an explicit target.

You can also use wildcards in the Composer's *Graph Data* dialog, as well as any raw URL query. We'll see examples of both of these later this chapter and throughout the rest of the book. Wildcards are an indispensable part of the Graphite experience; in many cases they allow you to perform queries that simply wouldn't be possible any other way.

The Graphite Composer window

It probably wouldn't shock you to hear that I ***love*** the Graphite Composer. There are some nice third-party apps out there for interacting with the Graphite render API and curating dashboards (looking at you, Grafana), but the Composer is still my go-to interface for any kind of serious troubleshooting. For my money, there isn't any better way to quickly prototype a graph to correlate and debug a production service or host that's gone "sideways".

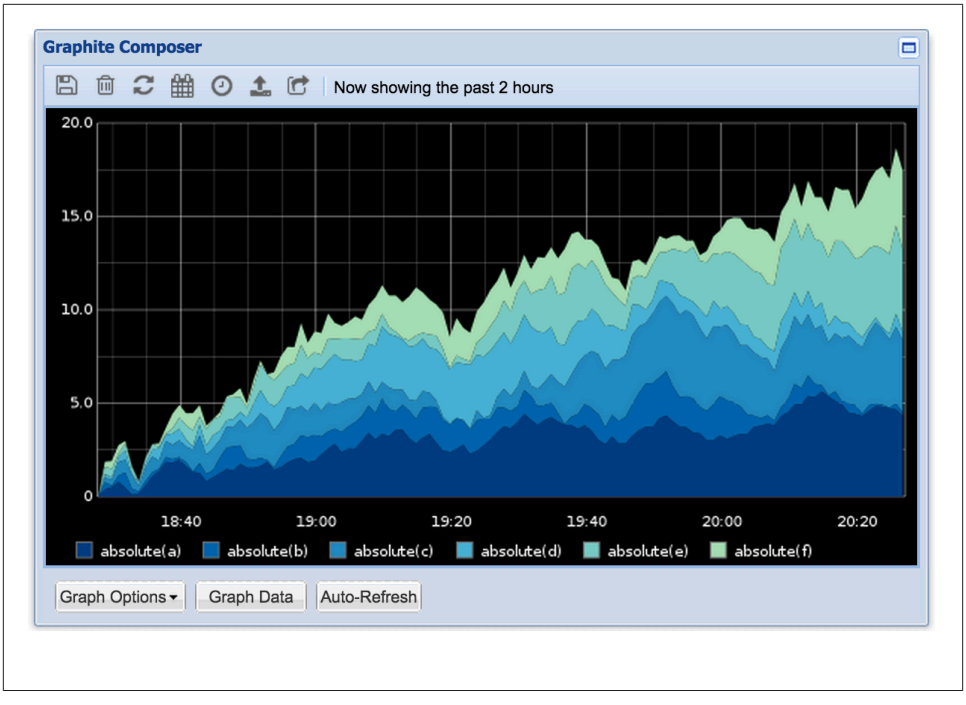


Figure 5-6. The Composer window

What makes the Composer such a useful tool? The distilled answer would be that it exposes the full power of Graphite's rendering API in a composition of UI buttons and menus, with *instant* visual feedback thanks to the embedded graph image. For every option you toggle or function you apply, you're immediately presented with the results of your change.

There are four primary functional elements that make up the Composer interface: the *Toolbar*, the *Graph Options* menu, the *Graph Data* dialog, and of course, the embedded *Chart*. Although the *Auto-Refresh* button stands on its own, I consider it a logical extension of the toolbar; it really has only one purpose - refreshing the chart every 60 seconds (and before you ask - no, the interval is not adjustable).

The Embedded Chart

While all the cool kids are doing client-side rendering of their graphs with JSON data and D3.js (which I'm a big fan of), there's something to be said for the straightforward *honesty* of rendering your graphs server-side with Graphite. Back when I was getting excited about monitoring software and graphs, and AJAX was just coming into its own, most developers were building their NOC displays and "walls of charts" with libraries based on rendering engines like Canvas and SVG. These technologies are fine for build-

ing client-side visualizations, but they have a moderate learning curve for the average Operations Engineer.

Contrast that with Graphite's default PNG output, which can be embedded in any HTML document with a basic `<img>` element (which is precisely how the Composer sources it, by the way). Suddenly you had the tools to construct complex analytical visualizations with a few clicks and an image URL. Because all of the parameters used to construct the graph were encoded in the URL, it could be easily adapted and automated with basic shell programming knowledge. Oh, and unlike its more *modern contemporaries*, you could actually download the PNG image store it for offline viewing.

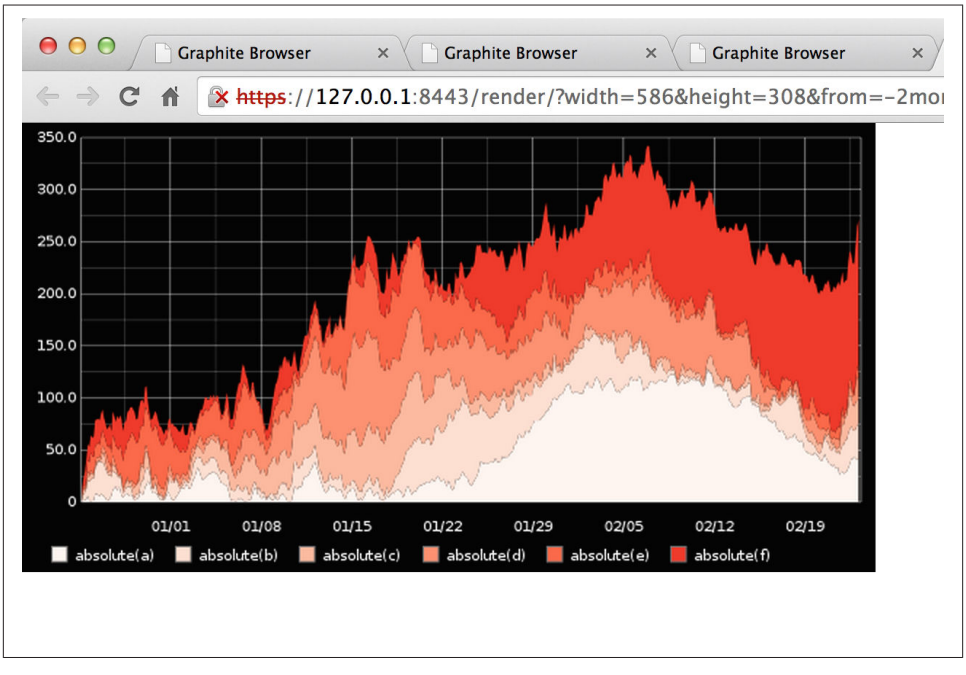


Figure 5-7. Loading a graph by URL

Graphite's PNG chart includes a number of useful primitives for you out-of-the-box: X and Y axes with dynamic tick values, a color-coded legend, and a grid consisting of major and minor lines. These can all be modified or disabled using the *Graph Options* menu, but they offer sane values by default.

Throughout the rest of this chapter we'll continue to refer to features by their UI labels, but I strongly encourage you to experiment and learn their associated API function names (and use) by right-clicking on the chart and opening it in a new browser tab. Studying the URL composition and playing around with the parameters is *absolutely*

the best way to master Graphite's rendering language. Case in point, Example 5-2 contains the URL (formatted for readability) used to generate the graph in Figure 5-7.

Example 5-2. The URL responsible for Figure 5-7

```
https://127.0.0.1:8443/render/
  ?width=586
  &height=308
  &from=-2months
  &areaMode=stacked
  &template=lava1
  &target=absolute(randomWalkFunction(%22a%22))
  &target=absolute(randomWalkFunction(%22b%22))
  &target=absolute(randomWalkFunction(%22c%22))
  &target=absolute(randomWalkFunction(%22d%22))
  &target=absolute(randomWalkFunction(%22e%22))
  &target=absolute(randomWalkFunction(%22f%22))
```

The Toolbar

Found in the upper-left corner of the Composer window, the toolbar is home to a collection of controls that manage the *state* of the graph, such as persistence (i.e. saving and deleting the graph) and the visible time range of data. When activated, almost all of these features expect input (or provide output) through an additional dialog. We're going to jump around the toolbar learning each of the features in a fashion similar to the way we might when debugging a service problem.

Note that the *Save* and *Delete* buttons are only visible if you're logged in to Graphite. All other buttons should be visible (and functional) since they don't require any user authentication.

Let's start off with a simple graph that we can use as the basis for monitoring the performance of Carbon and Whisper on your Graphite instance. Open your Graphite UI in your browser, expand the *Metrics Tree*, and drill down through your `carbon.agents` subfolders until you find the `metricsReceived` node. Click that metric to add it to your graph. Note that my metric path is **carbon.agents.synthesize-1.metricsReceived** although yours will likely be named something different.

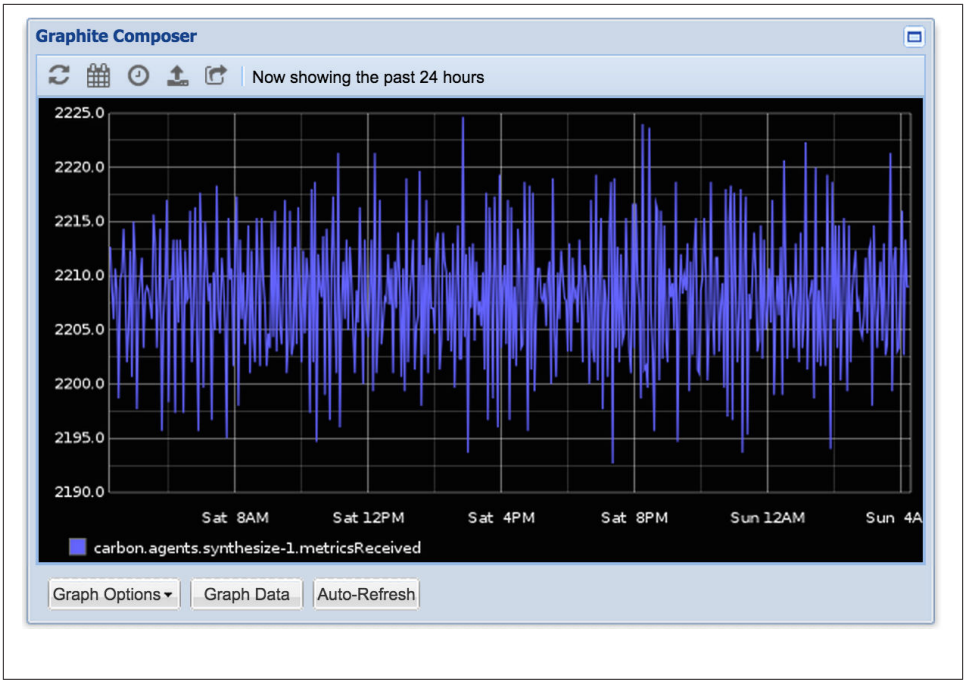


Figure 5-8. Building the initial graph

Your graph should look something like the one in Figure 5-8. Don't worry if your values vary greatly; my Graphite instance is also accepting metrics from `collectd`, resulting in a higher volume of *metrics received* by the `carbon-cache` listener (hint: this is what `metricsReceived` represents).

Selecting Recent Data

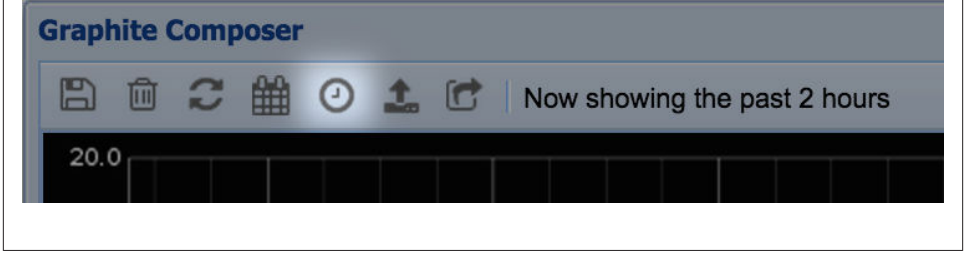


Figure 5-9. Selecting recent data

Anytime you start with a fresh Graphite window it defaults to a 24-hour view, relative to *this point in time*. This can often provide less context than we need since we're often

interested in *what's going on right now*, or at the very least, what's happened in the last few hours. We want to understand what **good** looks like before we try to determine when (and *how*) something **bad** happened.

The *Select Recent Data* button allows you to quickly adjust the time window by defining a quantity of time units (e.g. minutes, hours, days, etc). Let's update our view to show the last 30 minutes of data. If you're like me, you'll probably change the quantity field first from 24 to 30, and then move to the dropdown to change the units from hours to minutes. Note that as soon as your focus leaves the input field (i.e. when you click on the dropdown), the graph immediately refreshes; the Composer updates the graph every time a field loses focus. Make sure you complete the task by changing the unit to *minutes*.

Figure 5-10. Select last 30 minutes of data

Getting the Full Picture

A relative beginner might look at the data in Figure 5-8 as a seemingly chaotic series of datapoints. They jump between high and low extremes fairly frequently, at least within the range of visible data. This is actually a byproduct of Graphite's automatic scaling behavior; it zooms into a scale that makes it easier to analyze the given data for a par-

ticular view. If you set the `yMin()` (found under **Graph Options** > **Y-Axis** > **Minimum**) to a value of zero (0), you might discover that the data is rather predictable.

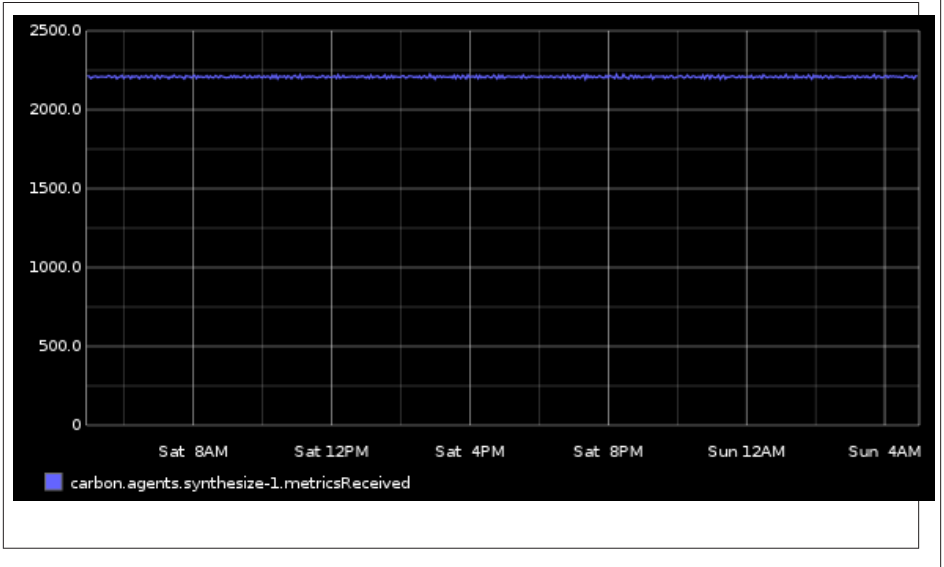


Figure 5-11. Set yMin for proper context

Refreshing the Graph

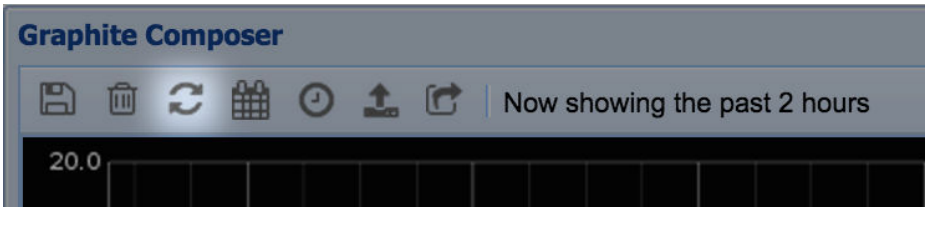


Figure 5-12. Refreshing the graph data

The *Update Graph* button refreshes the current view, as you might have already suspected. It should only have a noticeable effect when using relative time ranges (i.e. using *Select Recent Data*) after new datapoints have been received by your Graphite server. This isn't a hard and fast rule - you could have old or out-of-sequence data that arrived, affecting the view of a literal datetime range (explained in the next section), although this is far less common.

An alternative to using *Update Graph* manually would be to activate the *Auto-Refresh* button in the footer of the Composer window. This will cause Graphite to refresh the graph automatically every 60 seconds. Of course you can still continue to update the graph manually if you're in the middle of a situation where you need more frequent "real-time" updates. However, in this case, I generally prefer to load the graph in a dedicated tab with a browser extension to refresh the graph more frequently, e.g. every ten seconds. Yet another alternative would be a third-party dashboard with native functionality for updating the graphs more frequently, such as Tasseo or Grafana.

Figure 5-13. Auto-refreshing your graph

Selecting a Date Range

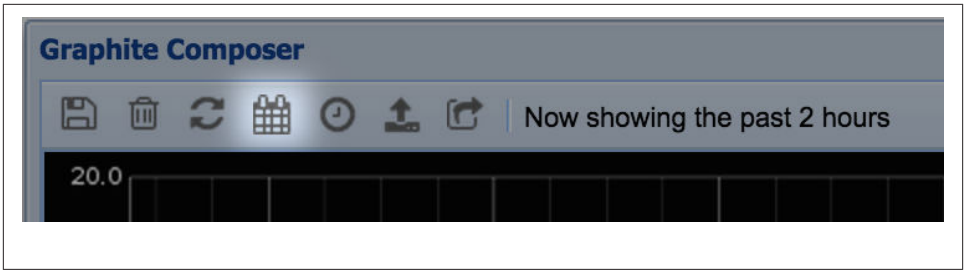


Figure 5-14. Selecting a date range

In the course of troubleshooting some (hypothetical) unusual behavior on my Graphite server, I added a couple new metrics to our graph: `committedPoints` and `updateOperations`. The former tracks the number of datapoints written to disk, while the latter represents the number of *update* operations performed across all Whisper files by that Carbon process. I noticed an anomaly from the usual stable behavior of these metric series, so I decided to zoom in and take a closer look using the *Select Date Range* dialog.

Clicking on the calendar icon will bring up a datetime selector, allowing you to select the start and end dates (and times) for the graph data. In my case I wanted to highlight a range showing a particular spike for `committedPoints`. Familiarize yourself with the various input methods for both month and hour selections.

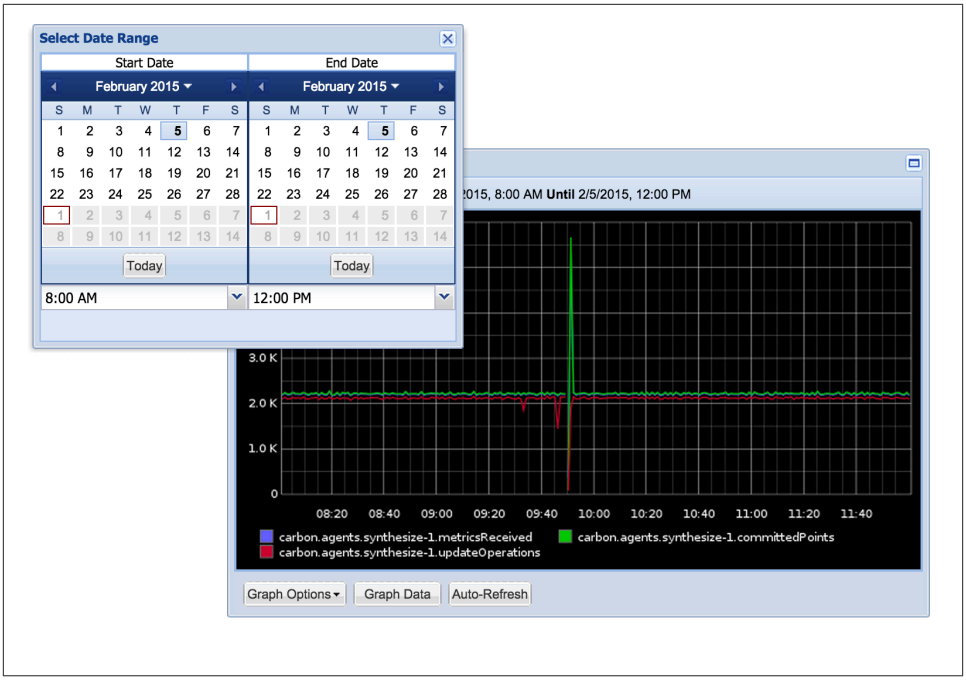


Figure 5-15. Zooming into a literal datetime range

If you find yourself jumping back and forth between "relative" (*Select Recent Data*) and "literal" (*Select Date Range*) time selections, it might not be immediately obvious that you can activate a previous selection by clicking the button, placing your mouse focus into one of the input fields, and pressing *enter*.

Exporting a Short URL

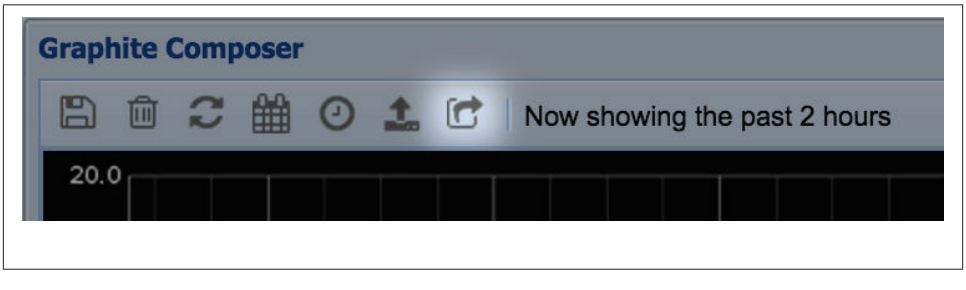


Figure 5-16. Creating a shortened URL

So we didn't find anything particularly interesting in that spike from the previous view, but to be extra vigilant, maybe we decide to share this graph with the rest of our team. I could right-click on the chart image and grab the URL, but our group chat service is one that truncates long messages, and we definitely want to avoid that.

Fortunately, Graphite has a feature to create a shortened URL from the current graph view. Clicking on the *Short URL* button will open a dialog with a URL that's convenient for sharing with anyone else that has access to your Graphite web interface. Note that this still only loads the graph itself; it doesn't load the graph *inside* of the Composer window, and these shortened URLs will **not** work directly within the *Create from URL* feature.

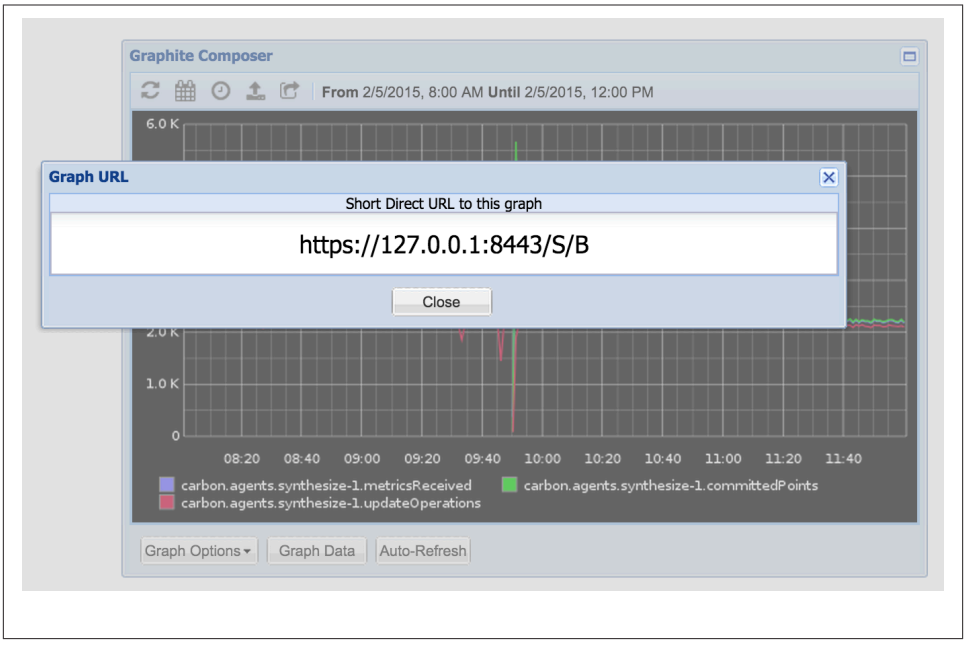


Figure 5-17. Sharing a graph with others

Loading a Graph from URL

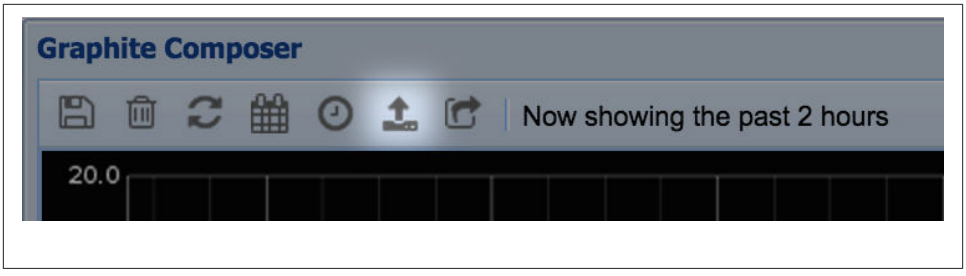


Figure 5-18. Loading a graph from an existing URL

The *Create from URL* button is useful for populating the Composer window with any existing Graphite chart URL. As mentioned in the previous section, it doesn't support a shortened URL directly since it only parses the URL; it doesn't have the capability to follow the HTTP redirection that occurs with shortened URLs.

It *does* support parsing all URL parameters, including time range and graph-level options, so the graph should look the same (relative time notwithstanding) for anyone loading the same URL.

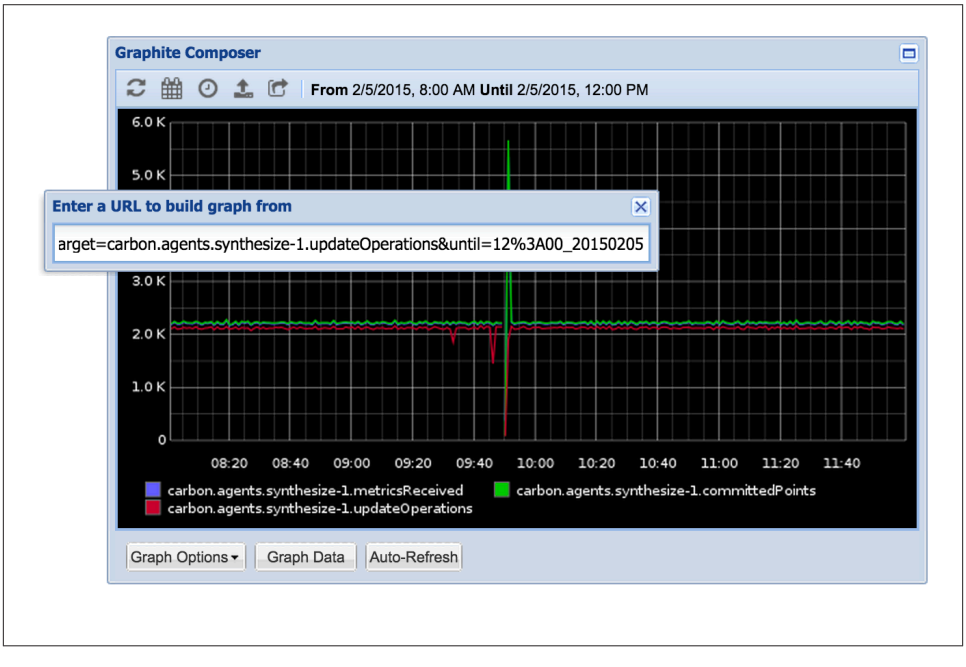


Figure 5-19. Loading a graph from an existing URL

Saving to My Graphs

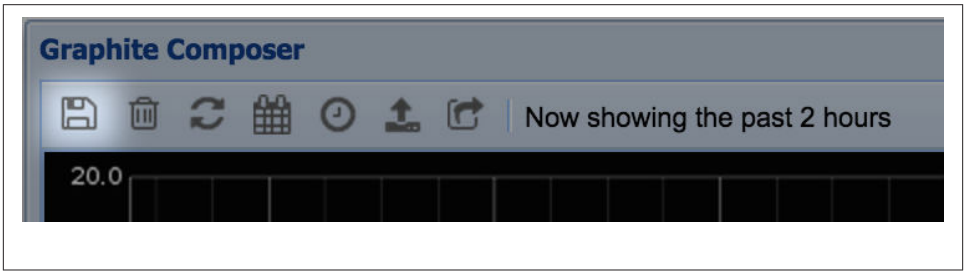


Figure 5-20. Saving to My Graphs

Assuming that this graph has some long-term value, we can save it in Graphite so that anyone on our team can retrieve it later. All saved graphs are visible under the *User Graphs* branch of the navigation tree. This is a handy way of reusing popular graphs across your entire organization, and is also a great way to share institutional knowledge about Graphite metrics and graph development patterns.

The *Save to My Graphs* button is only visible to authenticated users. Clicking on the icon will open a dialog for entering your graph name. Choose a descriptive but brief name that will convey the graph's purpose to all users. Here I've chosen to name our graph "Carbon_Performance" since we'll be using this (eventually) to monitor the health and performance of Carbon on this server.

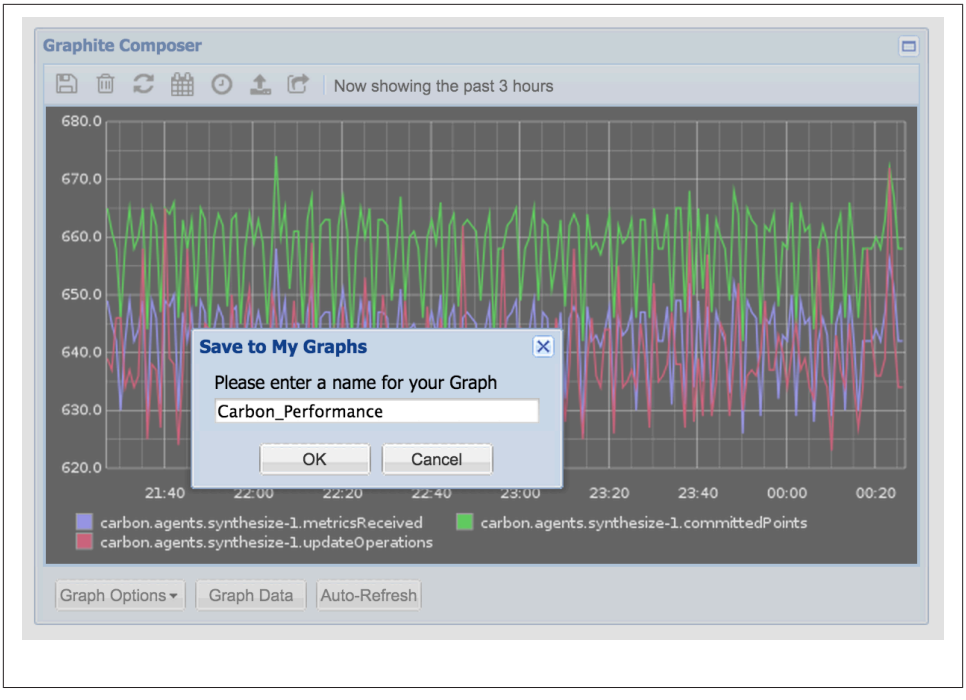


Figure 5-21. Saving a graph

Saved graphs can also be grouped and organized using the same dot-delimited hierarchy that Graphite supports for metric names. This makes it possible to categorize graphs into named folders in a structured manner.

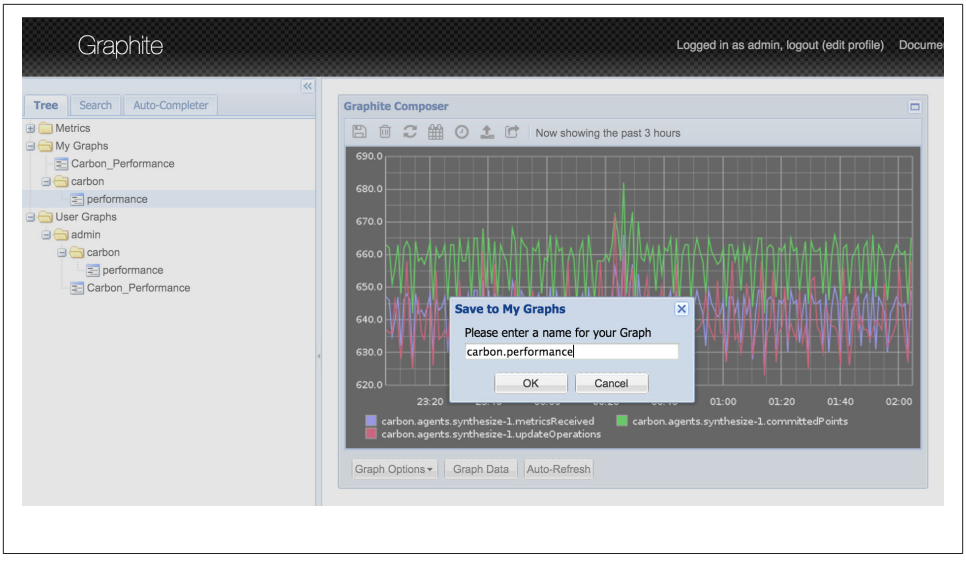


Figure 5-22. Saving graphs with a nested path

Deleting from My Graphs

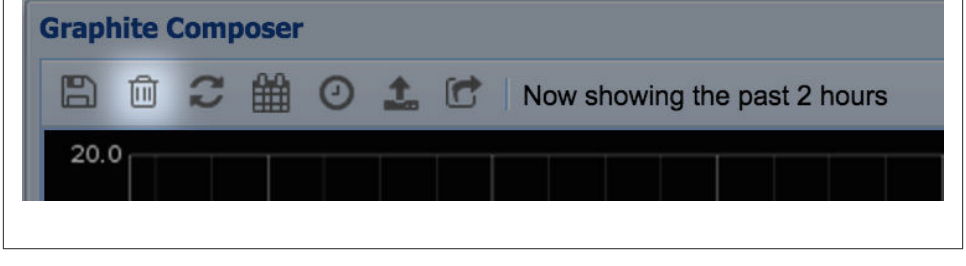


Figure 5-23. Deleting a graph

I really like the idea of organizing our graph under a collective **carbon** namespace since we're likely to need other related graphs, so let's delete the older **Carbon_Performance** instance we saved previously. Click the *Delete from My Graphs* button and a dialog will appear with the graph name to be deleted. Once you've taken a moment to verify that the name matches, go ahead and click *OK* to confirm the action.

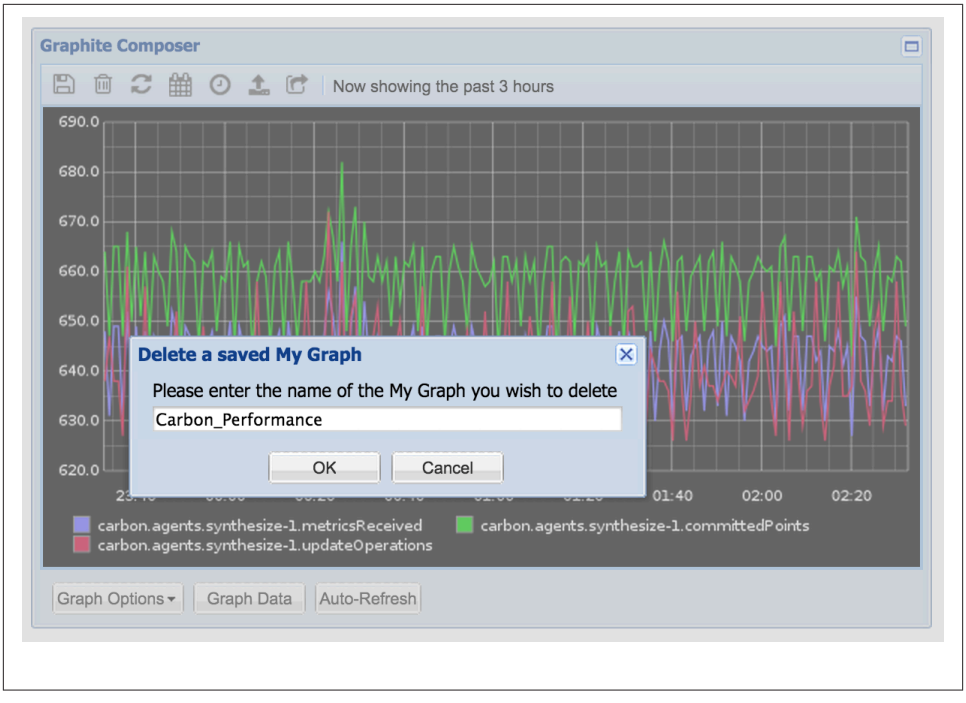


Figure 5-24. Deleting our previous graph

Note that this feature does **not** correctly populate the field when using nested (dot-delimited) graph names. If we were to try deleting the **carbon.performance** graph instead, only the last part of the name would appear in the field. You would need to prefix the field with **carbon.** before confirming.

The Graph Options menu

Graphite exposes a huge number of rendering options that affect the visual makeup of the graph itself, in contrast to the actual data rendered as lines (or as we'll see soon, as area shapes). These features are, not surprisingly, organized under the *Graph Options* menu button tucked away in the lower left-hand corner of the Composer window.

We're not going to cover every feature hidden in this menu, but we'll touch on the most frequently used ones (at least from my experience). The options that are obvious in nature, such as changing the background color, will be skipped over unceremoniously (almost certainly to be covered in the *second edition* of this book, if and when any such thing exists).

Pay special attention from here on out (as if you weren't already) as I'm going to include the actual rendering API function names as we discover each new feature. For example, we'll learn that the **Graph Title** entry in the menu sets the graph title, and uses the `title` parameter behind the scenes.

Adding a Graph Title

I like to think that a well-designed graph doesn't need a title in much the same way that good source code shouldn't need comments. Regardless, the Graphite developers provided us with a function to set our graph title. Let's go ahead and give our existing graph a `title` of **Carbon Performance**.

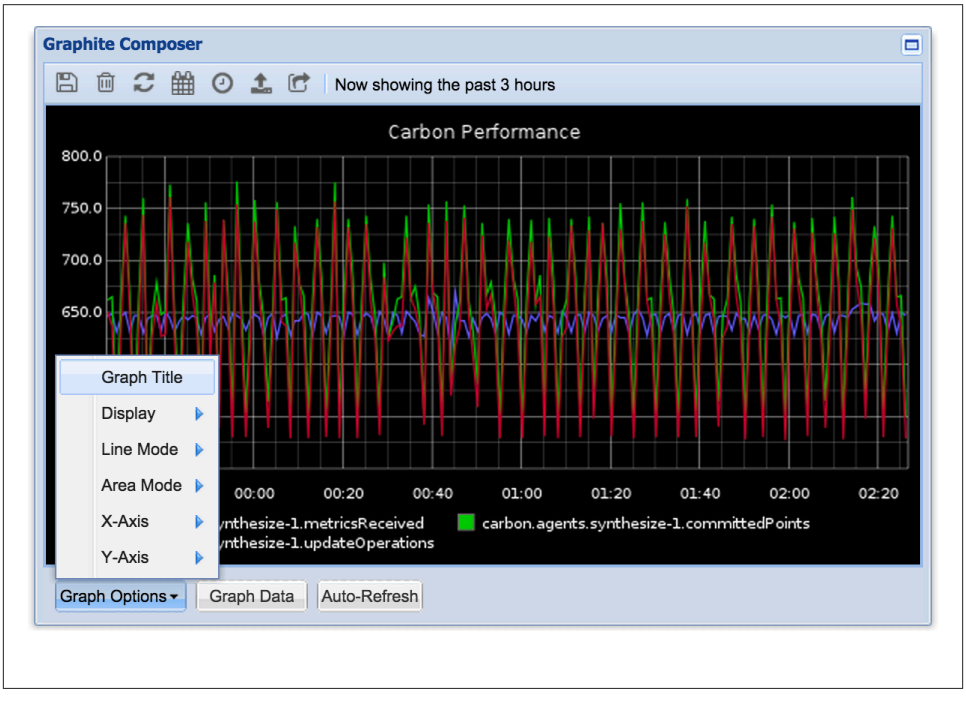


Figure 5-25. Set the graph title

Exercising your API skills

I stressed earlier that playing with the Graphite URL is the best way to learn the API, and I meant it. As we work through this chapter, take a few extra minutes to open the graph PNG in a new browser tab and experiment with the function parameters. Try changing the title to **Hello World** in a new chart without using the *Graph Options* menu!

Overriding the Graph Legend

The graph legend is clearly helpful but there may be times when you don't need it. Under the **Display** > **Graph Legend** submenu, there are a handful of settings that affect the visibility of the legend.

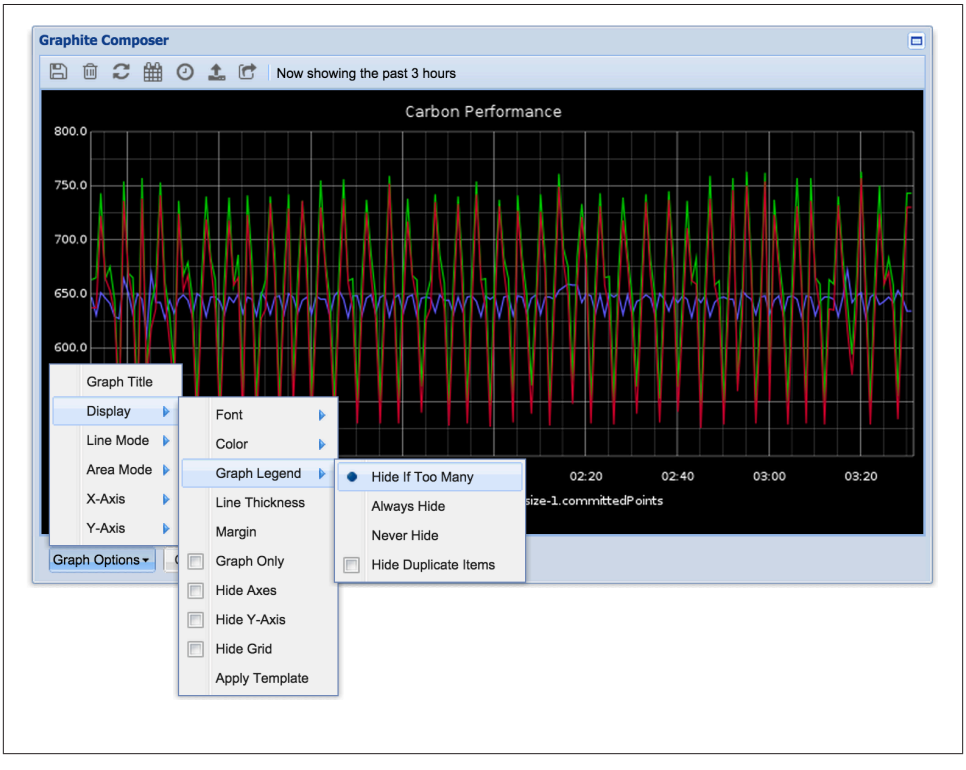


Figure 5-26. Graph legend settings

Hide If Too Many
: By default, Graphite will attempt to render the legend cleanly in the free space available. If there are too many entries to fit, the legend will be disabled. The absence of the other functions below implies the behavior in this selection.

Always Hide
: When `hideLegend=true`, the legend is disabled.

Never Hide
: When `hideLegend=false`, Graphite will render the legend at all costs. The vertical height of the graph will be extended to accommodate all legend items. This could

result in graphs that are difficult to manipulate around your visible screen, so use with caution.

Hide Duplicate Items
: When `uniqueLegend=true`, Graphite will consolidate duplicate legend entries. A *duplicate* must be a exact tuple match consisting of the entry *name*, *color*, and *axis* (left or right).

Toggling Axes and the Grid

Graphite renders a grid along with the X and Y axes to help make sense of the timeline and chart values. Each of these elements can be disabled individually, or altogether at once, using the checkboxes in the **Display** submenu.

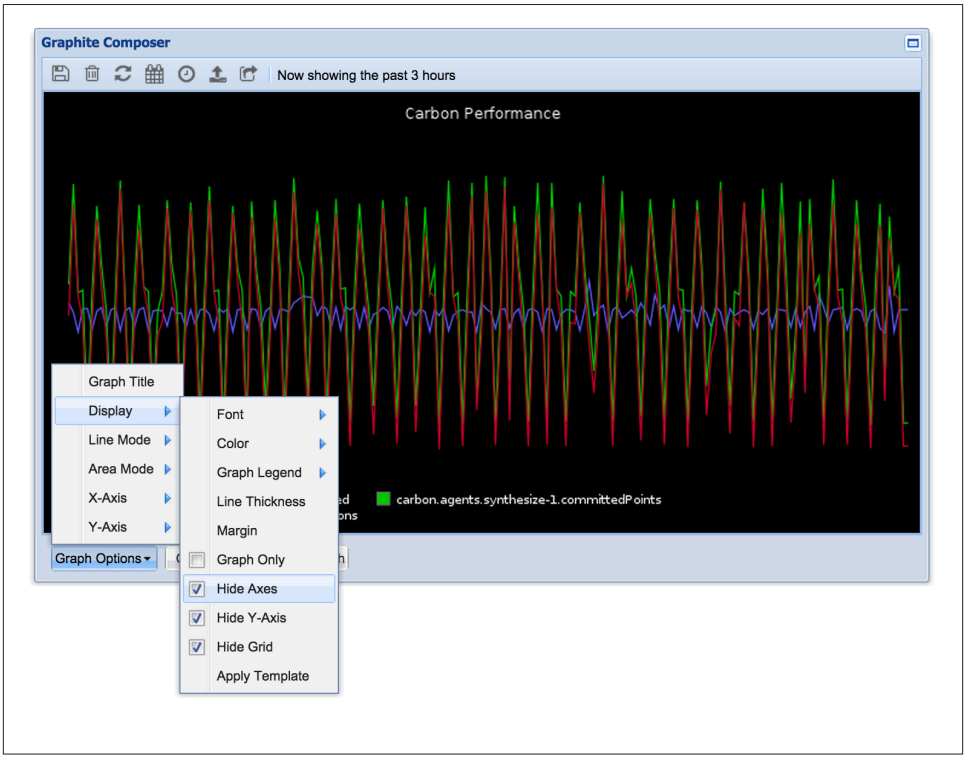


Figure 5-27. Axes and grid visibility settings

Graph Only
: This is what I affectionality refer to as the *nuclear option*. With `graphOnly=true`, only the data is visible as line or area series. All other elements, even the legend and title, are disabled.

Hide Axes
: This setting disables both the X and Y axes when `hideAxes=true`.

Hide Y-Axis
: When `hideYAxis=true`, only the X-axis is visible along the bottom of the graph.

Hide Grid
: When `hideGrid=true`, the major and minor gridlines are disabled in the background of the graph.

Applying a Graph Template

It's nice that we're able to customize so much of Graphite's visual experience, but it would be a real hassle if we had to type in our favorite settings every time we go to build a new graph. Fortunately, we can define preset *templates* in `graphTemplates.conf` containing a number of options for overriding font styles and color defaults.

Each template uses the same INI-style syntax we've seen before. The name of each template should be passed to the `template` parameter, which will load the preferences and apply them to your graph. There are no mandatory settings within each template definition. For example, those attractive graphs in Figures 5-6 and 5-7 were created using templates that *only* defined the `lineColors` setting.

Oddly enough, many of the settings in `graphTemplates.conf` do **not** map directly to their corresponding render API parameters, so here is a sample template with comments added to help clear up any possible confusion. Note that any parameters passed via the URL will override their related template settings.

Example 5-3. Sample visual template from graphTemplates.conf

```
[grafana]
background = white                              # bgcolor
foreground = black                              # fgcolor
minorLine = grey                                # minorGridLineColor
majorLine = rose                                # majorGridLineColor
lineColors = #7eb26d,#eab839,#6ed0e0,#ef843c    # colorList
fontName = Sans                                 # fontName
fontSize = 10                                   # fontSize
fontBold = False                                # fontBold
fontItalic = False                              # fontItalic
```

Line Chart Modes

Although it doesn't support interpolation or fancy-shmancy curve smoothing, Graphite gets the job done when it comes to rendering line charts. There are a couple modes that affect the shape of the line, and a few more choices that address *nulls* (i.e. gaps) in our data. All of these settings are found under the **Line Mode** submenu. Each of the line

modes (`slope`, `staircase`, and `connected`) are passed to the API using the `lineMode` parameter.

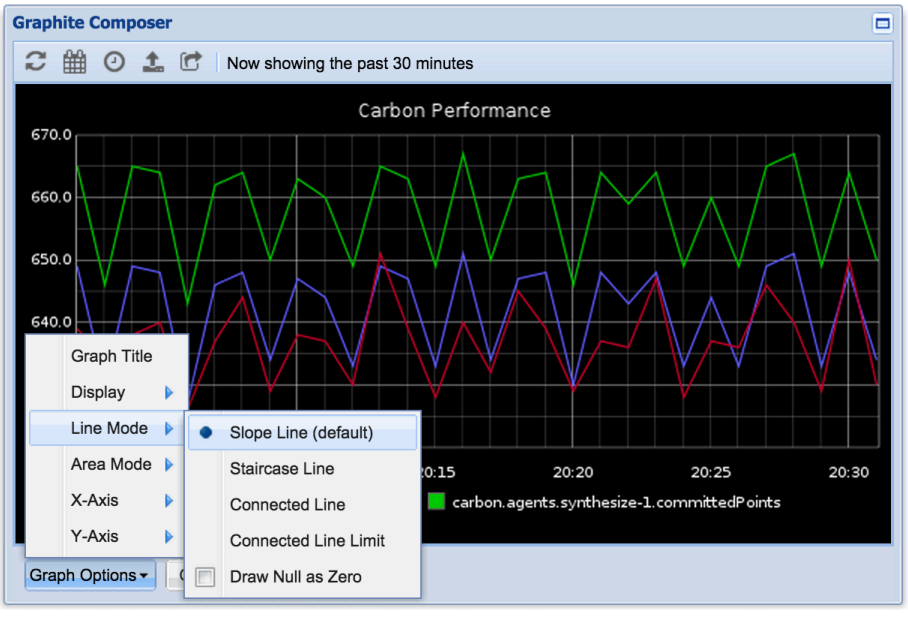


Figure 5-28. Line chart modes

What is Interpolation?

In layperson's speak, and within the context of time-series data, *interpolation* is a way to backfill missing datapoints in a set of data. It's possible for measurements to get lost in transit, resulting in gaps in our graph. Interpolation is used by some tools to replace these missing datapoints.

Graphite offers a couple alternatives to interpolation. First, the *Connected Line* mode will draw a line directly between known datapoints on either side of a gap. Note that this is only a visual effect available when rendering an actual image; it has no effect on actual data series you might encounter with, for example, *raw* or *json* output formats. Second, the `keepLastValue()` function can be used to repeat the *last known value* until a non-null value is available.

Slope Line

This is Graphite's default render type and the one used most commonly throughout this book (and in real life). The `slope` line connects each consecutive datapoint but does **not** span gaps caused by missing or null datapoints. The graph in Figure 5-29 is not an *attractive* version of a slope line chart per se, but it clearly demonstrates the effect of gaps in your data.

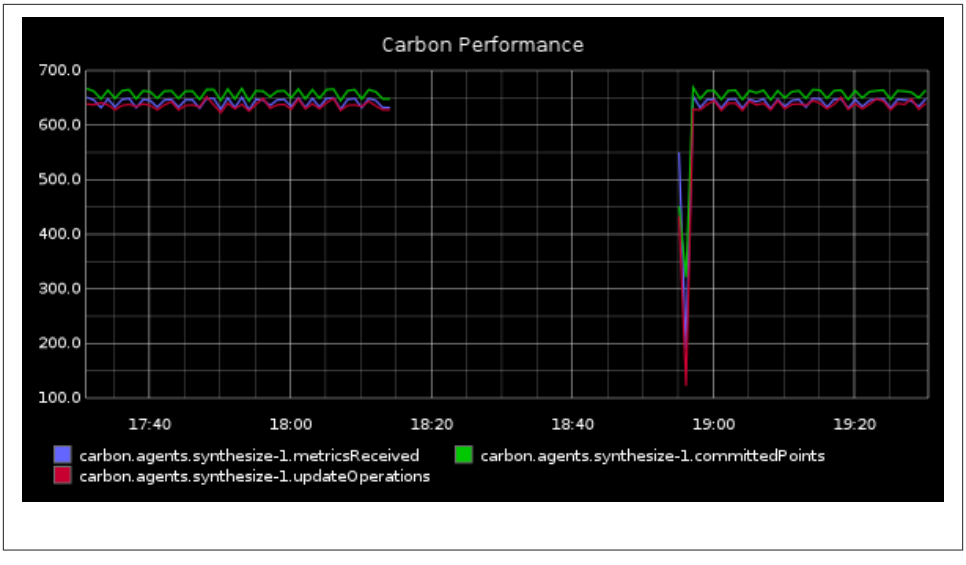


Figure 5-29. Line chart using the default slope

Staircase Line

Specifying `lineMode=staircase` renders a stair-stepped line chart instead of the usual slope line. Each "step" illustrates the boundaries of the interval for the datapoint, making it particularly handy for viewing summarized or downsampled data. Personally, I like to use the staircase line in combination with *stacked area* charts to create a *slightly janky* version of a histogram.

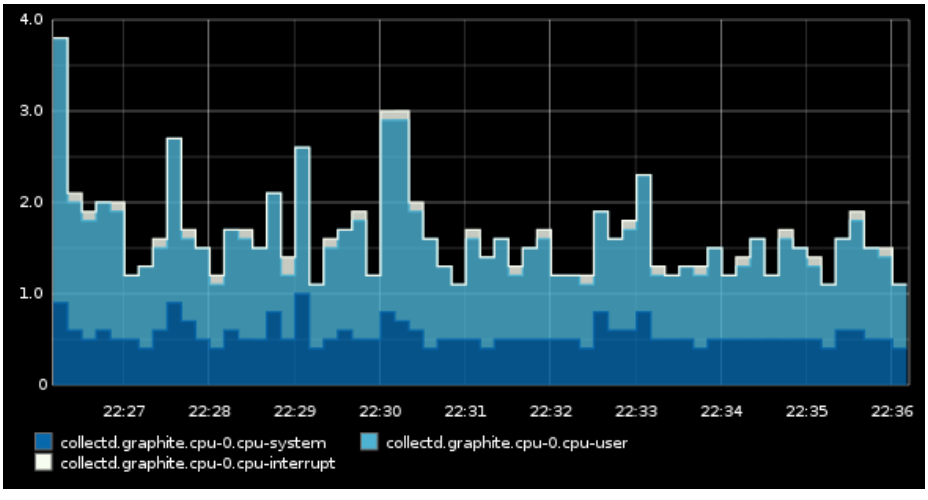


Figure 5-30. Stair-stepped line chart

Connected Line

The Graphite user's version of "connect the dots", `lineMode=connected` spans gaps in your chart to connect the available datapoints. Although I've already mentioned this, it probably bears repeating that this mode does **not** use interpolation.

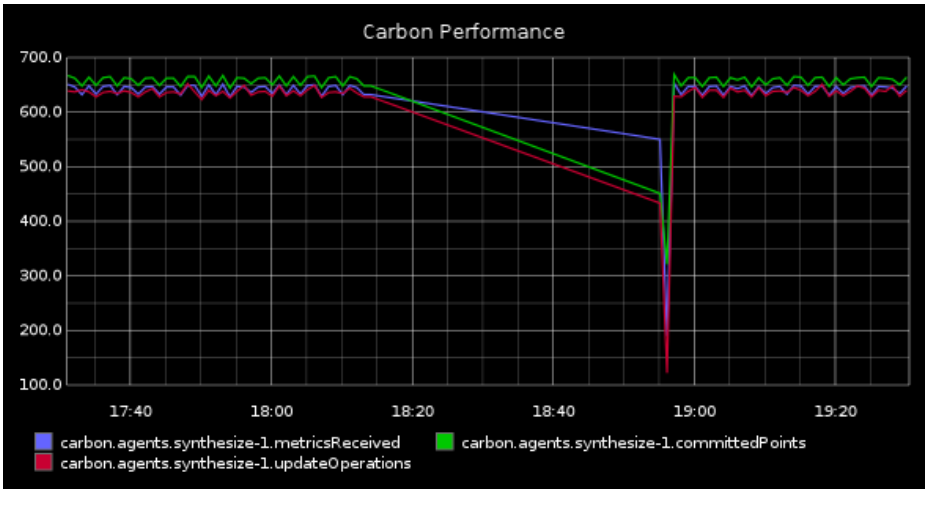


Figure 5-31. Using connected lines over gaps

Connected Line Limit

The `connectedLimit()` function is supplemental to *Connected Line* graphs. When enabled, the user will be asked for a numeric value which specifies the maximum number of *consecutive null values* to allow. If a gap contains too many null values it will simply refuse to "span the gap" with a connected line.

Draw Null as Zero

When `drawNullAsZero=true`, Graphite converts all null datapoints to zeroes. This can be particularly useful if you're combining multiple series, such as adding two series together with `sumSeries()`.

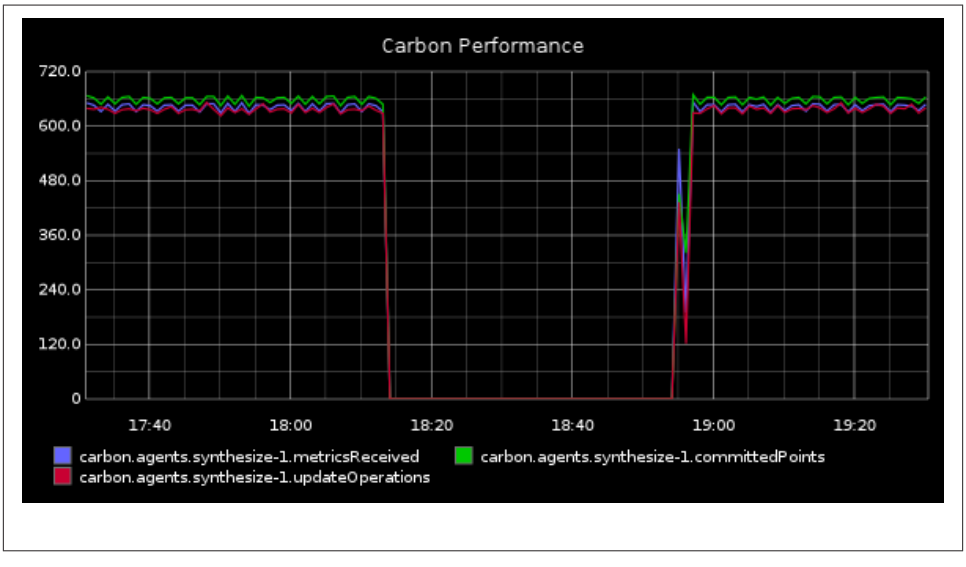


Figure 5-32. Convert nulls into zeroes

Area and Stacked Graphs

Area charts are an effective way to compare and contrast different series against one another. They're also helpful for visualizing the aggregate of a collection of metrics, such as how the different states of memory on a server add up to the whole. Graphite provides us with a few different modes for rendering data as two-dimensional regions: `first`, `stacked`, and `all`. These are all located within the **Area Mode** submenu.

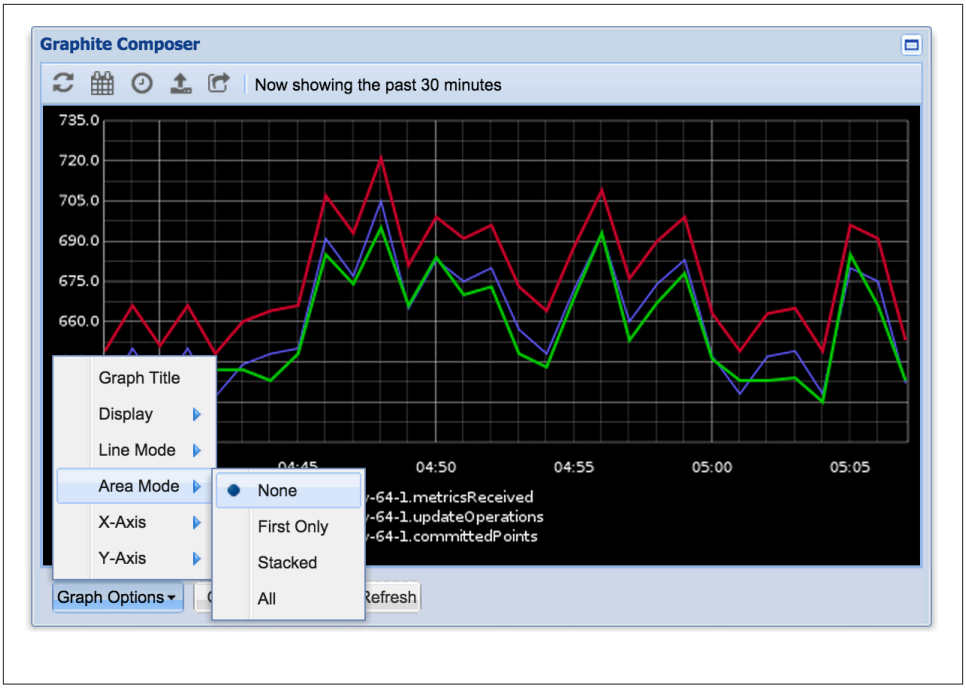


Figure 5-33. Area chart modes

First Only

Our *first* area mode (pun definitely intended) is `areaMode=first`, which causes Graphite to render only the first listed target as an area series. All other targets will be drawn as standard line series. I find this mode particularly useful for highlighting one metric as a baseline, or merely for emphasis, to contrast against the remaining series.

Figure 5-34. Area mode for first target only

Stacked

As its name implies, `areaMode=stacked` allows us to create graphs of stacked area series. This type of chart has a variety of use cases: observing discrete metrics that combine to form a whole, such as system memory (see Figure 5-35); tracking aggregate use of a common metric across a class of systems; or similarly, identifying *anomalies* within a common metric across a class of systems. Observing the seasonality of an aggregate metric is significantly easier using stacked graphs than trying to do the same with a line chart.

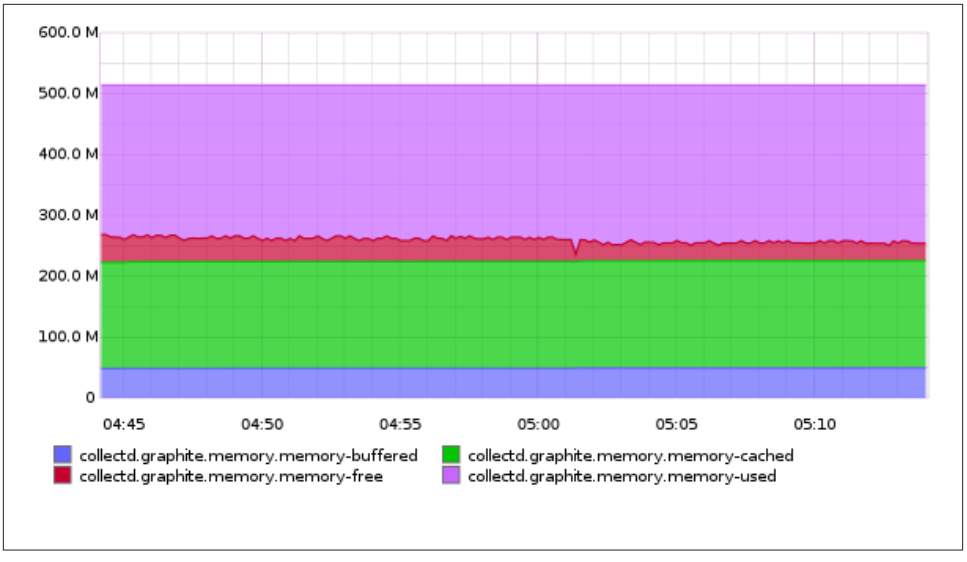


Figure 5-35. Stacked memory metrics

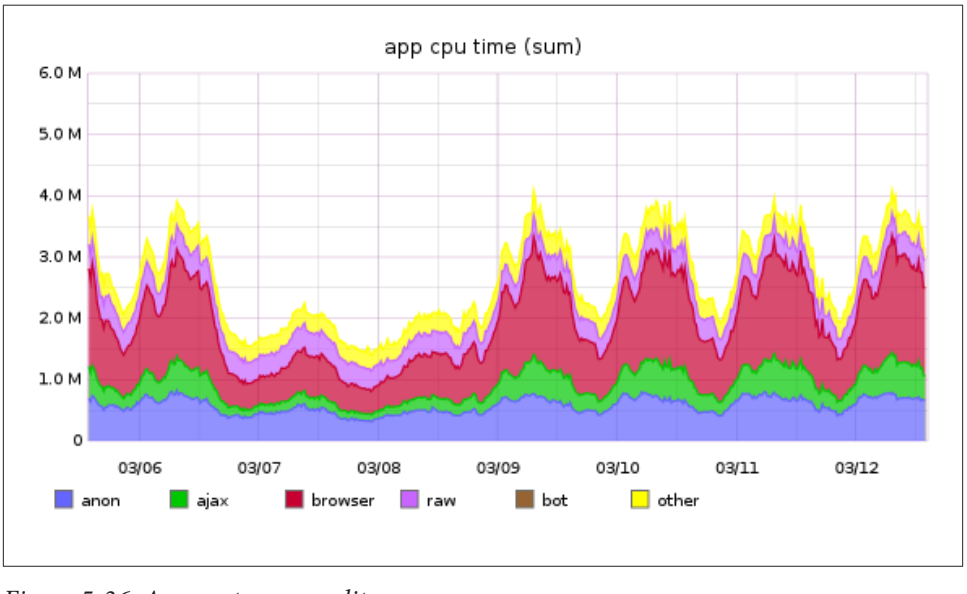


Figure 5-36. Aggregate seasonality

All

Our final choice, `areaMode=all`, renders all of the targets as area series. Each subsequent series appears "in front" of the one preceding it, which can result in some confusing

visuals. I would only recommend this mode when each series diverges enough to make it visually distinct from its neighboring targets. It can also be helpful to activate *Filled Area Alpha Value* (see below) to increase the transparency of each area region.

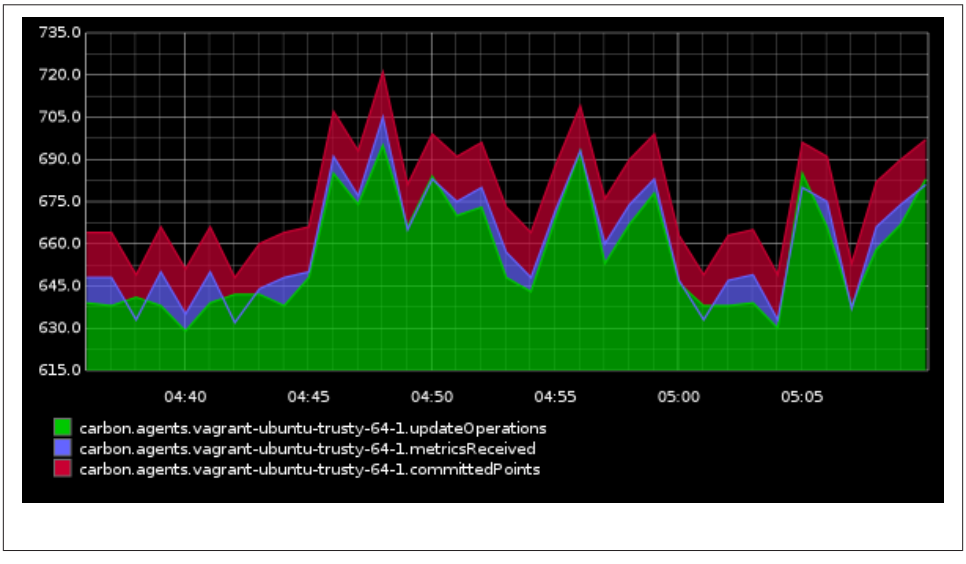


Figure 5-37. Non-stacked area for all targets

Filled Area Alpha Value

Using the default color palette, Graphite area charts have been known to sear retinas. Fortunately, there is a setting, *Filled Area Alpha Value*, hidden deep under the **Display** > **Color** submenu that allows us to control the level of alpha transparency for colors in area graphs. It accepts a decimal value between `0.0` (full transparency) and `1.0` (total opacity).

Each of the area and stacked graph examples in this section used `areaAlpha=0.7` for a more pleasing visual effect.

You can also control the alpha transparency for each area series individually using the `alpha()` function. However, this function does **not** override the graph-level `areaAlpha` parameter. In order to use `alpha()` for one or more series, you would need to disable `areaAlpha` first.

We haven't covered per-series functions yet so don't worry, you didn't miss anything. These are coming up in the *Graph Data* section just a bit further ahead.

Tweaking the Y-Axis

If you read my earlier chapters, you might recall that I place a *lot* of emphasis on the ability to compare seemingly disparate datasets to discover correlations, such as the effect of network latency on "widgets sold". In order to study these metrics side-by-side, sometimes it helps to calibrate our Y-axis so that each series is maximized within the scope of the visible graph. Graphite includes a variety of settings under the **Y-Axis** submenu that offer this sort of flexibility.

A Not-So-Quick Example

Mastery of the Y-axis is one of those skills that pays off your investment in effort many times over. You could say that about a lot of Graphite skills, but understanding how to manipulate the axes is especially helpful for troubleshooting real problems in production. Let's look at a contrived example using system metrics gathered by `collectd`.

My boss, let's call him "Lumbergh", asked me to check into a momentary spike in memory used on one of our servers (in this case, the Graphite server itself). On a whim, I look to see if there was any unusual activity in disk writes. At first glance, Figure 5-38 seems to show that I'm probably on a wild goose chase.

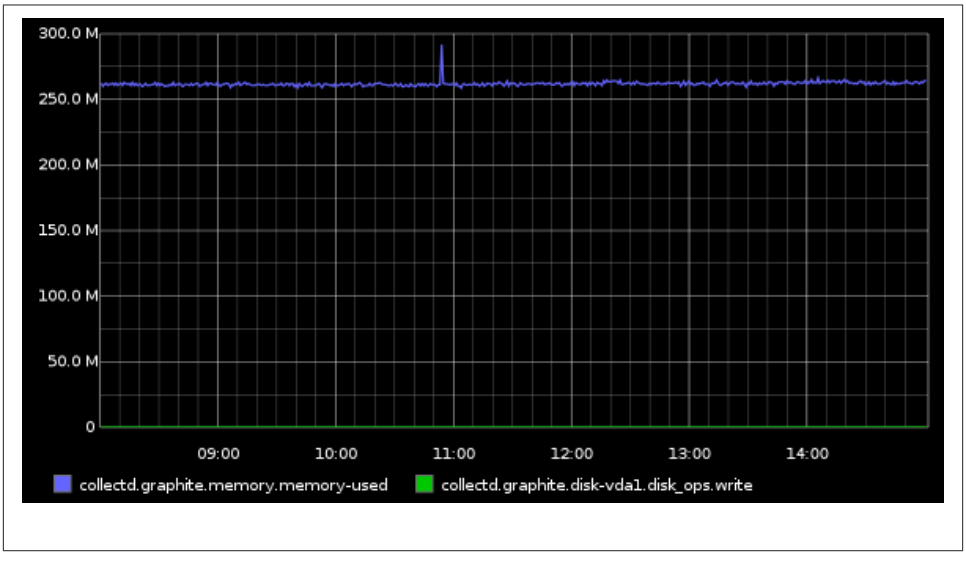


Figure 5-38. Comparing memory use and disk writes

However, I'm familiar enough with these metrics to understand that their values are way too far apart to compare accurately. In order to accommodate *bytes of memory used* on our Y-axis, the graph's scale is totally out of whack for the number of *disk ops written* — it looks like there's no activity at all!

Fortunately, we can dissociate the two metrics by moving either series to the *second Y-axis*. As demonstrated in Figure 5-39, the *Graph Data* dialog gives us the ability to enhance and transform series by applying per-series rendering functions to each one independently. Click the **Graph Data** button in the bottom of the Composer window to open up the list of targets. Once you have the window open, click on the series you want to modify and then choose the **Apply Function** > **Special** > **Draw in Second Y Axis** function.

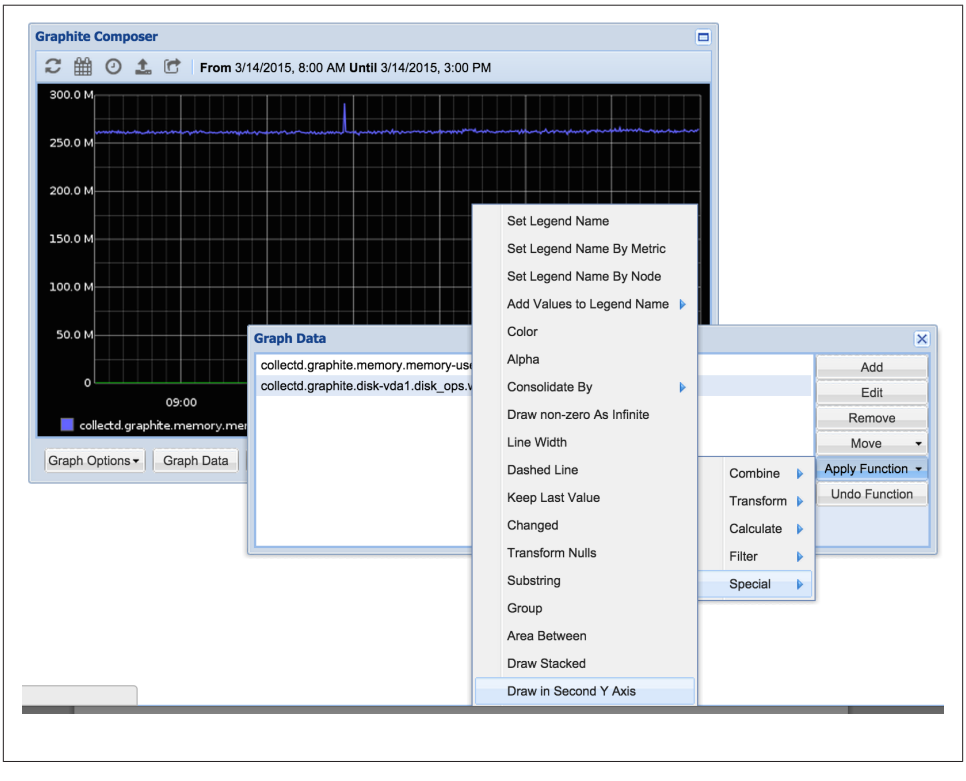


Figure 5-39. Moving disk metric to Right Y-axis

As you can see from the legend in Figure 5-40, our `disk_ops.write` metric is now wrapped by the **`secondYAxis()`** function. This is now associated with the *Right Y-Axis*, which has its own scale, units, etc. But more importantly, it's immediately obvious that there **is** a correlation between these two metrics. Now, there's nothing here to prove that this is a *causal relationship*, but we haven't ruled it out either. This is at least progress.

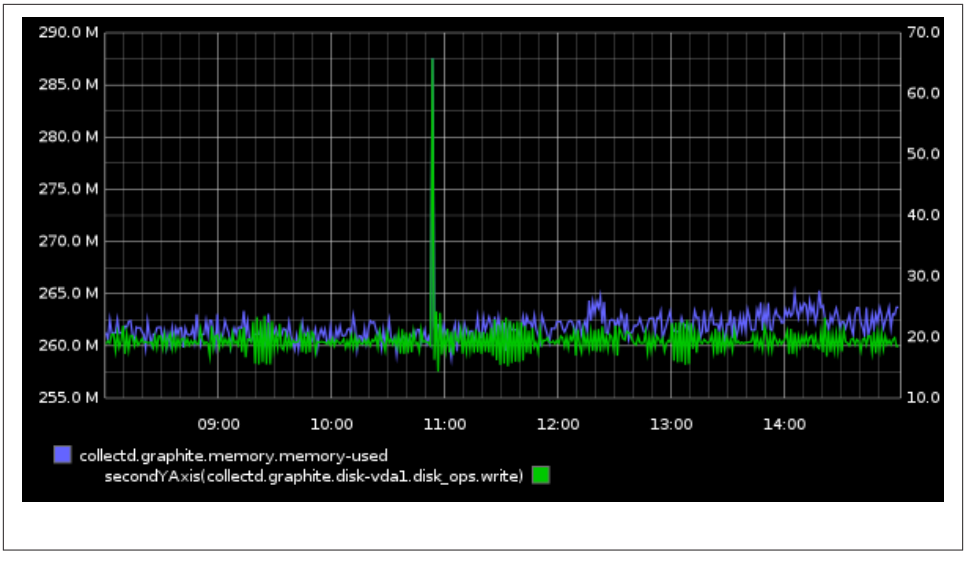


Figure 5-40. Metrics on either Y-axis

Of course, if the answer was really this easy, Mr. Lumbergh probably wouldn't have interrupted our TPS report training session. There's still one dilemma here: we can see the related spike in disk activity, but now it's obfuscating our memory spike! What's a ladder-climbing Graphite engineer to do?

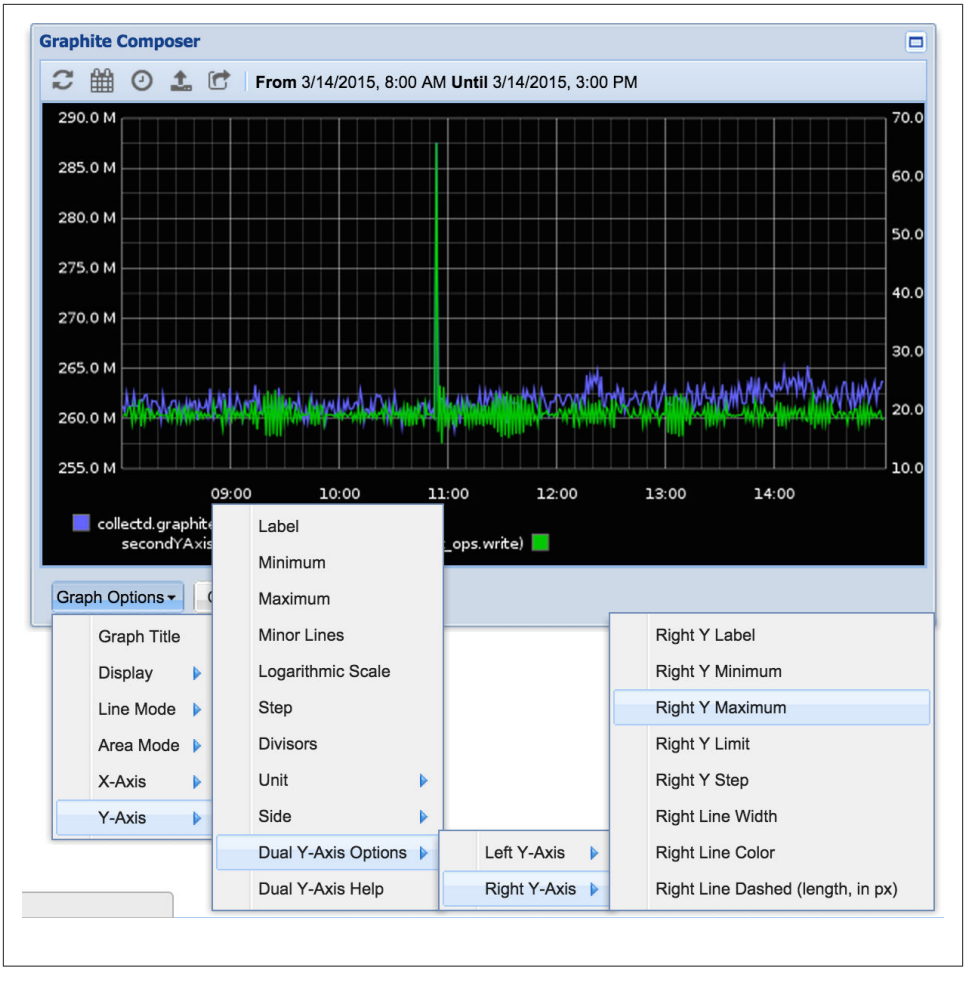


Figure 5-41. Changing the Right Y-axis upper boundary

Glad you asked. We still have another trick up our sleeve. In it's ultimate wisdom (well, that of the programmers who built it), Graphite wants to help us out by adjusting each Y-axis scale so we can view as much as possible of the vertical range as possible; unfortunately, this is getting in our way. We can outsmart Graphite by setting the upper boundary of the axis artificially high; this should "smoosh" the line series down far enough that both spikes will be visible.

Let's make this change using **Right Y Maximum** under the **Y-Axis** > **Dual Y-Axis Options** > **Right Y-Axis** submenu. For our purposes I'll enter **200** at the prompt for our upper boundary.

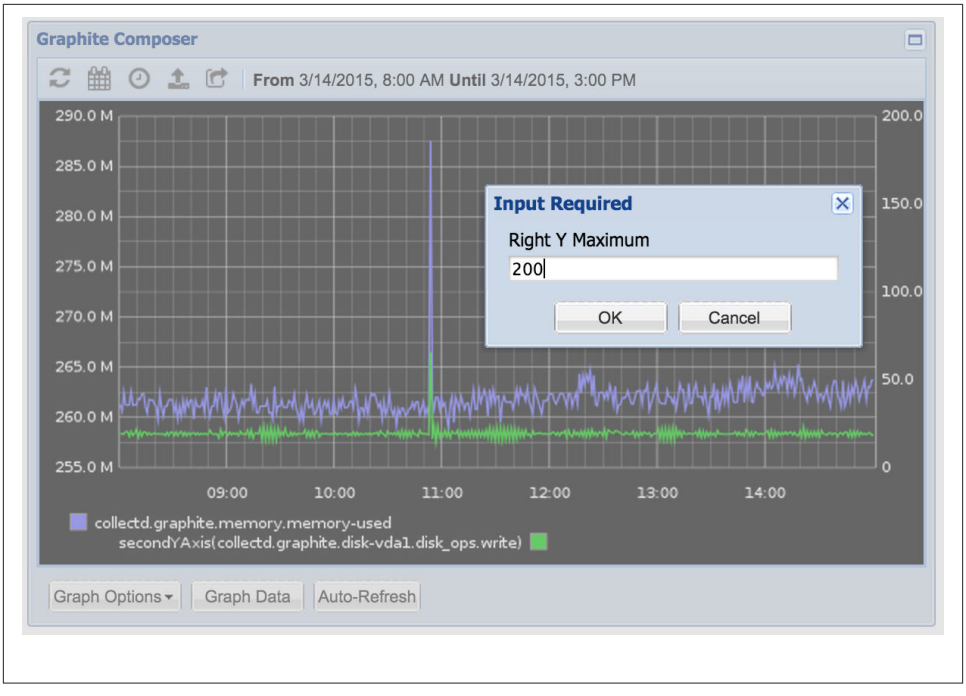


Figure 5-42. Set yMaxRight to 200

Both spikes are now clearly visible, demonstrating correlation if not causation. Figure 5-43 reveals a graph that I'd be proud to present to Mr. Lumbergh. You might even say this graph has *flair*.

Terrible puns aside, I think you can appreciate the usefulness of dual Y-axes and the ability to manipulate the scale and boundaries of each one independently. My apologies for revealing the *Graph Data* window this early, but I think it's important to acknowledge Graphite's **`secondYAxis()`** support when discussing the various Y-axis features.

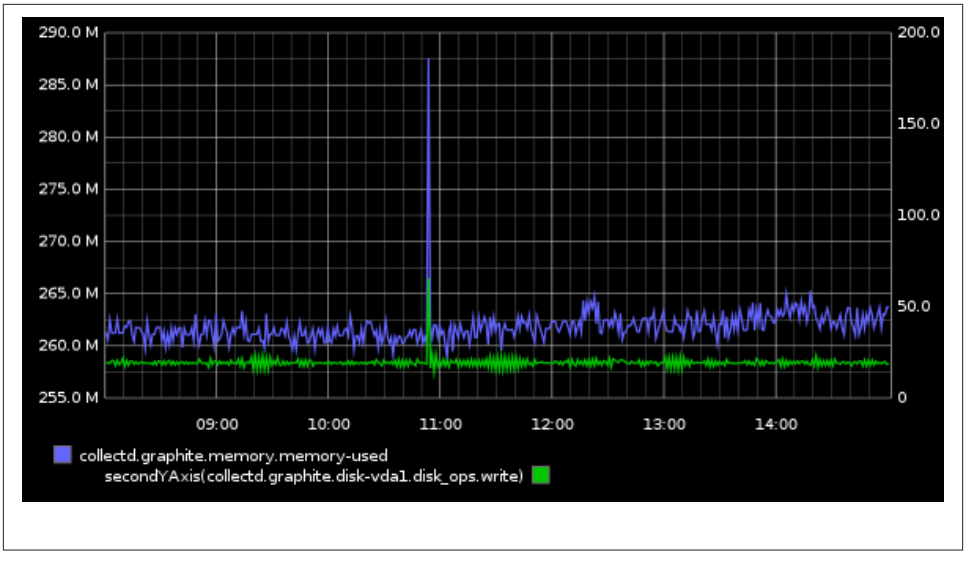


Figure 5-43. Revealing both spikes

Y-Axis Labels

Graphite supports *labels* on each Y-axis. This is an arbitrary string, although most folks use it to specify the metric units or to classify a collection of metrics aligned on the same axis. Take Figure 5-44, which uses a lot of the same tactics I demonstrated in the previous example. It's a clear and concise graph expressing the bi-directional traffic on a network interface. But would it be obvious that each of the targets has been modified with **`scaleToSeconds()`** if I hadn't labeled each Y-axis accordingly?

Coincidentally, the left and right Y-axis labels are set with the API parameters `vtitle` and `vtitleRight`, respectively.

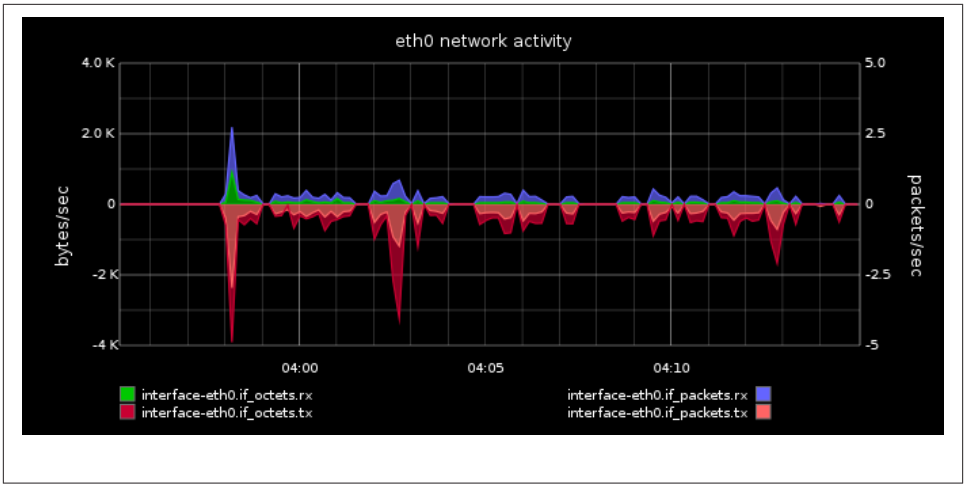


Figure 5-44. A good case for using labels

A note about Dual Y-Axis Options

Most of the settings found under the **Y-Axis** > **Dual Y-Axis Options** submenu are *only* applicable when the Right Y-Axis (`secondYAxis()`) is being used. It's easy to fall into the habit of using **Dual Y-Axis Options** if you're constructing a lot of complex, dual-axis graphs. Before long you're beating your head against the wall trying to figure out why `leftDashed` or `leftWidth` isn't having any effect on your graph.

Or so I hear. *From a friend.*

Upper and Lower Boundaries

For completeness, and because we're getting *perilously* close to the *Advanced Graphing* chapter, this feels like a good time to start revealing the constructed URLs responsible for these sample graphs. Example 5-4 is a formatted version of the URL I used to build the previous graph, reinforcing many of the features and concepts we've learned in this chapter. In particular, pay note to the parameters that control the upper (`yMaxLeft` and `yMaxRight`) and lower (`yMinLeft` and `yMinRight`) boundaries. Of course, if you're dealing with a single Y-axis, you can get by just fine with `yMax` and `yMin`.

In this example, and many of the subsequent ones throughout the book, I've taken to formatting the **`target`** parameters with indention, just as you might with a programming language like Python, Perl, or Ruby. I find this makes it much easier to *digest* the structure and meaning of complex *target* definitions. Functions and their secondary arguments should appear with the same indentation level, with their primary argument (almost always the *metric* or another function name) appearing at the next indentation

level. If you're like me, you'll probably find that it's easiest to read these starting at the metric "root" and working backwards, just as the graph creator would have, adding functions one at a time.

You might also notice some encoded strings in there, such as the `vtitle`. In practice these graph "resource locators" are completely *url-encoded*, converting whitespace and other characters into a corresponding *unreserved character*. I've decoded many of the values below to make the example easier to read, but left them intact for title and label strings.

Example 5-4. A reasonably elaborate graph

```
https://127.0.0.1:8443/render/?width=680&height=308
  &vtitle=bytes%2Fsec
  &vtitleRight=packets%2Fsec
  &yMinLeft=-4000
  &yMaxLeft=4000
  &yMinRight=-5
  &yMaxRight=5
  &areaMode=all
  &areaAlpha=0.7
  &title=eth0%20network%20activity
  &target=
    substr(
      scaleToSeconds(
        color(
          collectd.graphite.interface-eth0.if_octets.rx
        ,"green")
      ,1)
    ,2)
  &target=
    substr(
      scaleToSeconds(
        color(
          secondYAxis(
            collectd.graphite.interface-eth0.if_packets.rx
          )
        ,"blue")
      ,1)
    ,2)
  &target=
    substr(
      scaleToSeconds(
        color(
          secondYAxis(
            scale(
              collectd.graphite.interface-eth0.if_packets.tx
            ,-1)
          )
        ,"pink")
      ,1)
```

```
  ,2)
&target=
  substr(
    scaleToSeconds(
      color(
        scale(
          collectd.graphite.interface-eth0.if_octets.tx
        ,-1)
      ,"red")
    ,1)
  ,2)
```

I would **love** to be able to print these `target` parameters in their full horizontal glory, but with traditional book sizes being what they are, this simply isn't feasible. I tried to convince my editor that a shorter, wider publication would be Totally Fine™, but they insisted that it be retitled to *Graphite: The Coffee Table Book*.

The Graph Data dialog

I'm really glad that you've made it this far. Besides the fact that we're that much closer to the next chapter (where we start *really* ramping up your graph skills), the *Graph Data* dialog is one of my favorite parts of the Graphite Composer. Graphite is nothing without metrics, and this little dialog is where we get to play around with metrics and functions, transforming them from mundane datasets into *beautiful* and functional portals of information.

Earlier we peeked at the functionality of the *Graph Data* interface, using it to add metrics to the Right Y-Axis. Until that point, we'd worked with simple charts and graph-level options, without any sort of data transformation or filtering. From here on out we expose the full power of Graphite *targets*, building on simple metrics and elevating them into something much more useful, and combining them to form charts that tell a story or solve a problem.

What are Targets anyways?

What makes the *Graph Data* dialog unique, in contrast to the *Toolbar* and *Graph Options* menu, is that we're dealing specifically with the data series that will eventually be rendered as lines or areas on our chart. Even if every other graph-rendering feature was literally ripped out, we'd still be left with something to analyze.

This interface is designed to simplify the work of constructing and customizing complex *target* definitions that describe the data in our graphs. As you might recall, every render URL that describes a Graphite graph consists of one or more targets and some supplementary graph parameters. Each target defines a metric (or multiple metrics, in the case

of wildcards), along with an optional sequence of functions intended to combine, filter, transform, or otherwise manipulate the data (or its label), serially.

Let's consider one of the targets from the previous section.

```
&target=
  substr(
    scaleToSeconds(
      color(
        scale(
          collectd.graphite.interface-eth0.if_octets.tx
        ,-1)
      ,"red")
    ,1)
  ,2)
```

Walking through each sequence, we can follow along with my thinking as I constructed the target.

1. I want to add the metric **`collectd.graphite.interface-eth0.if_octets.tx`** to our graph. It measures the number of octets (or *bytes*) being transmitted (`tx`) outbound on interface `eth0`, which is something I happened to need to know at the time.
2. One trick for creating more usable space on a graph is to use the negative Y axes. By applying **`scale(-1)`** to this metric, Graphite will render the series below the X-axis. This has the added benefit of mirroring the *receive* metrics (`rx`) on the positive Y-axis.
3. I choose to render this particular metric with **`color("red")`** for reasons which will forever remain shrouded in mystery.
4. Remembering that my collectd agent reports every 10 seconds, but that network activity is typically reported in *per-second* intervals, I apply **`scaleToSeconds(1)`** to handle the conversion.
5. Finally, I use **`substr(2)`** to set a "pretty" label, removing the nonessential bits of the metric name. Here it grabs everything starting from the node at the 2nd index (`interface-eth0`), and continues to the end of the string (`tx`), since a terminating index is not provided.

The Graphite rendering API has beautifully concise syntax, but even an expert would quickly tire of writing these definitions from scratch. With the Graph Data dialog we can quickly iterate through our target expressions with a largely point-and-click interface, without the mental gymnastics of composing each sequence manually.

In the next section we'll step through the process of building our own "Carbon Performance" graph with custom targets as we learn the features of the Graph Data dialog.

The end result is a chart that you'll want in your Graphite toolbox for monitoring and debugging your own Carbon servers.

Building a Carbon Performance graph

Elsewhere in this book I've highlighted the importance of Carbon's internal statistics and how valuable they are when it comes to managing the health and performance of your Graphite servers. The one thing I neglected to tell you is exactly *how* to organize and read these metrics effectively. We're going to fix that oversight now using what we've learned this chapter, in addition to picking up some knowledge about the *Graph Data* dialog.

Adding metric targets

If you haven't already, let's start this exercise with a fresh Composer window. You can either reload your existing Graphite window, or load the Graphite UI in a new browser tab. When the interface is ready, activate the targets list window with the **Graph Data** button, and then click on the **Add** button to open the *Add a new Graph Target* modal.

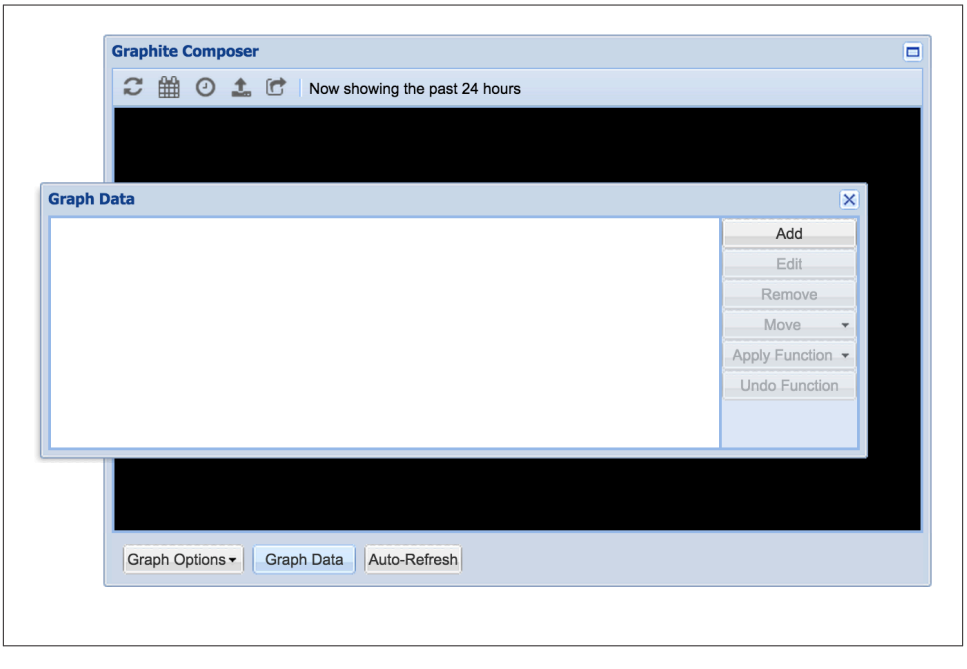


Figure 5-45. The Graph Data dialog

One of the first things you may notice is that the modal interfaces for adding and editing targets (the *Add* and *Edit* buttons, respectively) use the same interactive elements that

we discovered in the *Auto-Completer* metrics navigation tab. This means that we can search for metrics using auto-completion and incorporate wildcards in our metric string definitions.

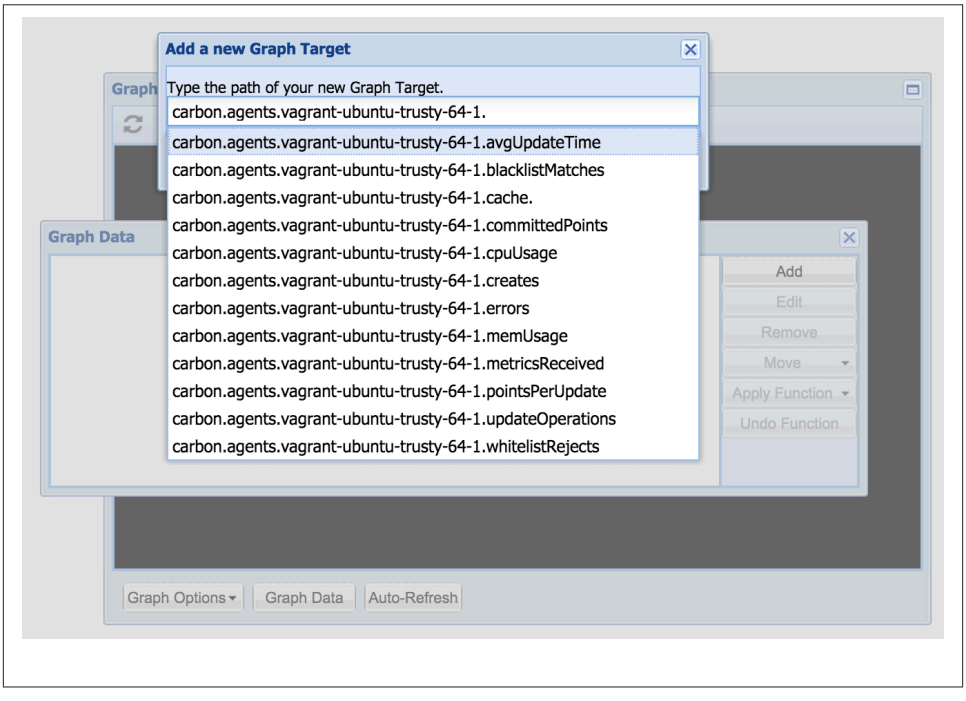


Figure 5-46. Adding targets directly

Although your exact metric path may differ slightly from the one in my examples (your Carbon hostname is probably different), you shouldn't have any problem tracking down the location of your `carbon-cache` statistics. You should have a directory (or "node") with the same list of metrics as those in Figure 5-46. Once you find them, proceed by adding each of the following metrics to your target list.

- **`metricsReceived`**: number of metrics received at the `carbon-cache` listener
- **`updateOperations`**: number of Whisper update operations performed
- **`committedPoints`**: number of Whisper datapoints written to disk
- **`cpuUsage`**: combined *user* and *system* CPU time used by the `carbon-cache` process
- **`pointsPerUpdate`**: number of batched datapoints written per update operation
- **`creates`**: number of new Whisper files created

Each of these metrics are contextually specific to the process that created them, e.g. `carbon.agents.vagrant-ubuntu-trusty-64-1.creates` records the number of new Whisper files created by the cache process that was started with `--instance=1` on host *vagrant-ubuntu-trusty-64*.

For a more exhaustive study of these metrics (and more like them), check out the *Troubleshooting* chapter where we learn about these statistics and the myriad logfiles produced by the Graphite components.

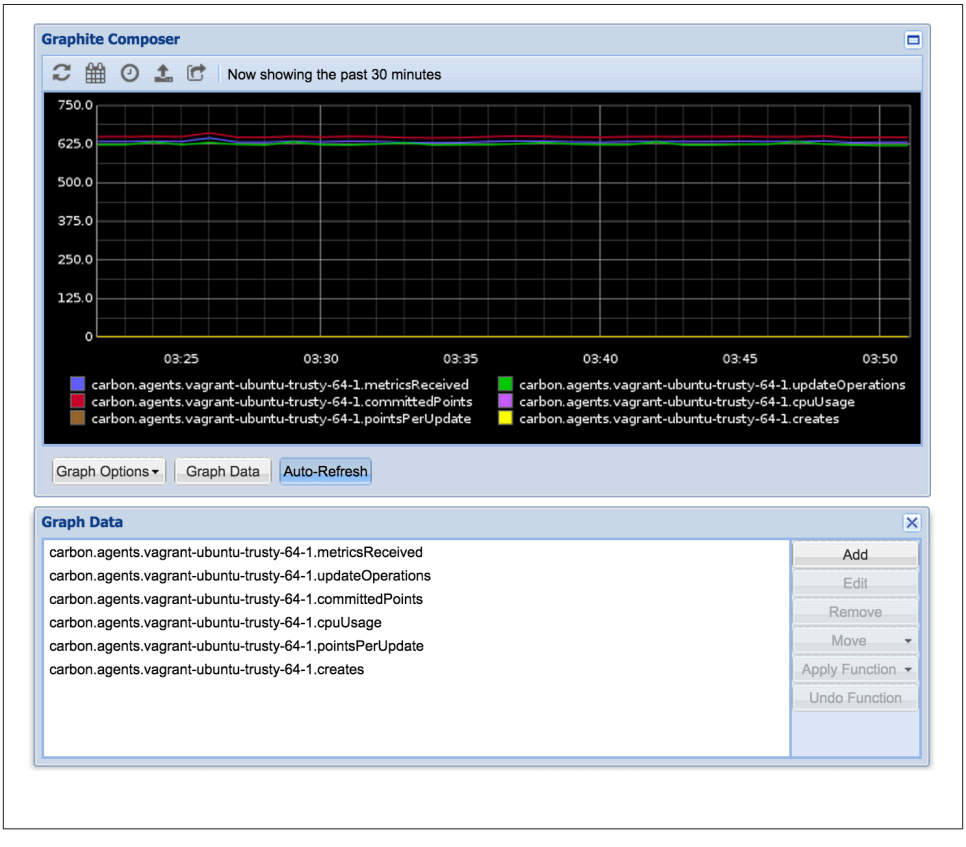


Figure 5-47. Carbon metrics to monitor

Adding functions to targets

Assuming you have active metrics, your chart should look something like the one in Figure 5-47. The first obvious problem here is that some of our metrics are on wildly different scales. At a glance we can see that `metricsReceived`, `updateOperations`, and `committedPoints` all have values hovering around 625.0, with the rest settled at the base of the graph around zero (0).

Because we have two nicely grouped strata, we can move one of the sets to the second Y-Axis. I tend to move the group with the lower values to the secondary axis, but this is completely arbitrary and subject to personal preference.

If you're following my lead, go ahead and select the `cpuUsage`, `pointsPerUpdate`, and `creates` targets so that all of them show "selected" with a light-blue shading. Add the `secondYAxis()` function to these targets by activating **Apply Function** > **Special** > **Draw in Second Y Axis**.

Selecting multiple targets at once

The target list will let you perform actions on multiple targets at once. This dialog supports the common multiple-select actions that you're probably familiar with from computer user interfaces and web forms: Mac owners will use the *command* key while their Windows and Linux counterparts will use the *control* key.

You should also be ok to use the *shift* key for a contiguous selection, but I generally avoid this because browsers tend to also treat the action as if you're highlighting the text *within* the selection. This doesn't really cause any errant behavior, but it's an *unpleasant* interraction, in my experience.

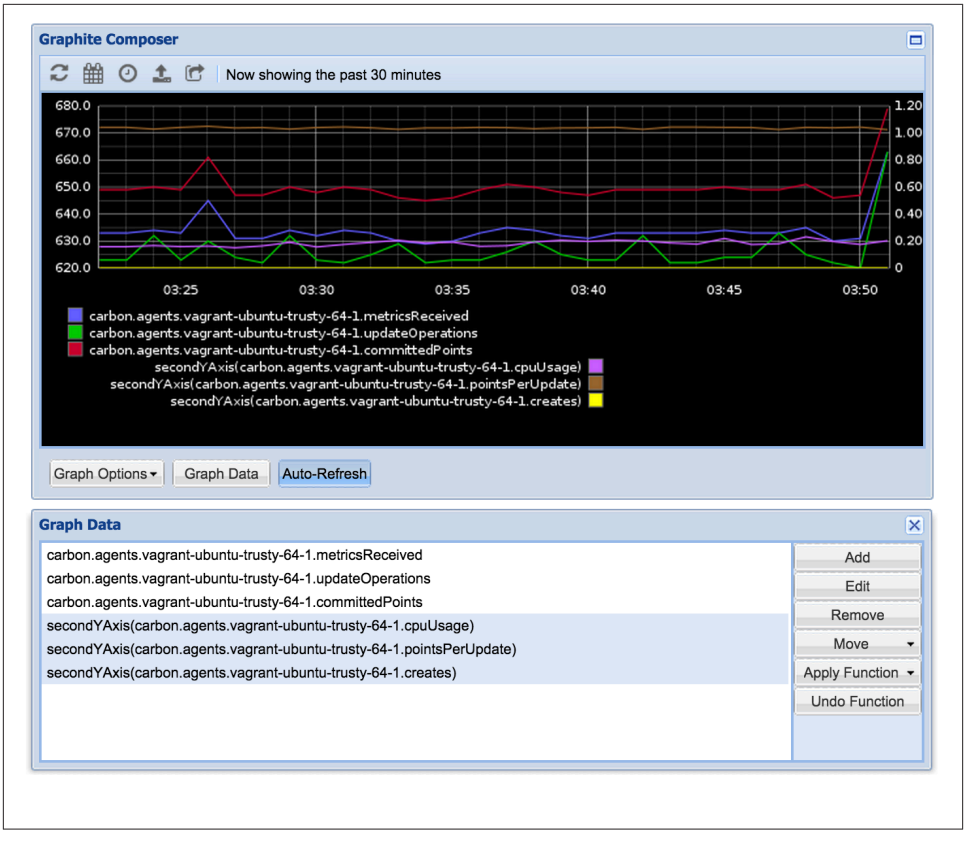


Figure 5-48. Using multi-select to apply functions

As expected, moving those targets to the second Y-Axis allowed Graphite to zoom into a narrower range of values, resulting in a much more "readable" graph. However, the lengthier target definitions are also fully visible, causing some weird alignment issues in the legend. Fortunately, the render API includes a variety of *aliasing* functions designed to modify the visible target string in the legend.

Select the entire target list and then choose **Apply Function** > **Special** > **Substring**. Use a value of **3** for the node index ("starting position"). Your result should look similar to the graph in Figure 5-49.

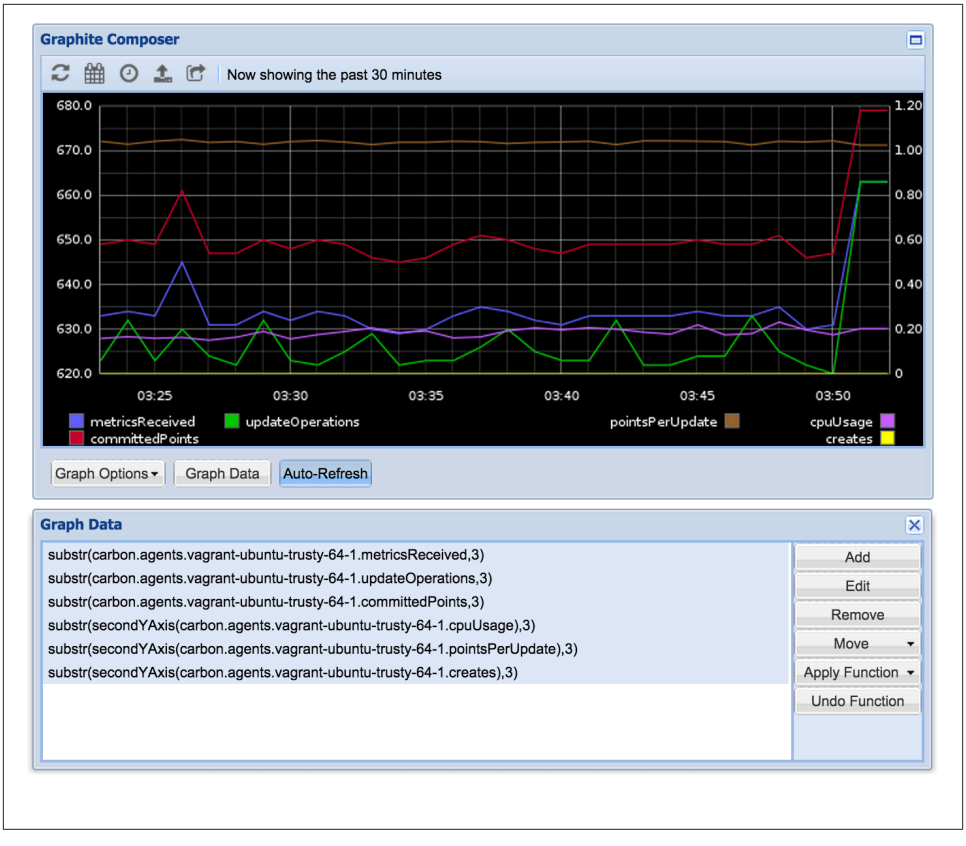


Figure 5-49. Functions that change the label

Just like you might with network traffic, it's good idea to get in the habit of reporting your Carbon traffic in terms of *metrics per second*. The Carbon daemons report their internal statistics every minute, so let's employ `scaleToSeconds()` here to adjust the rate for each target. Select the `cpuUsage`, `pointsPerUpdate`, and `creates` targets and activate **Apply Function** > **Transform** > **ScaleToSeconds**. Enter a value of **1** at the prompt.

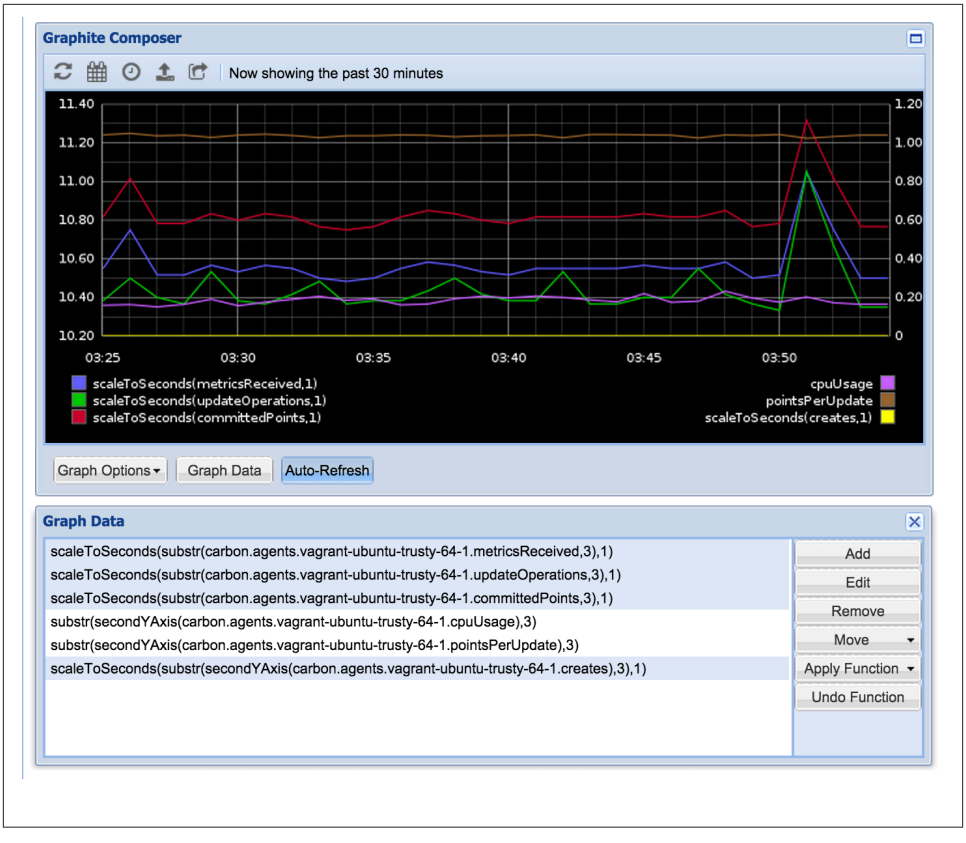


Figure 5-50. Adjusting the rate for a subset of targets

Removing functions from targets

The graph in Figure 5-50 looks really close to our desired result. Unfortunately I made a bone-headed mistake and applied `scaleToSeconds()` **after** `substr()`. As you can see in the legend, the order of functions applied to a target really does matter.

I won't go so far as to say that I'm anal-retentive about my legend labels, but we'll treat this as an opportunity to learn the *Undo Function* button. Activating this widget will remove the outer-most function from each target currently selected in the target list window. Use this feature to remove the `scaleToSeconds()` and `substr()` functions and reapply them to each affected target, in the proper order. Your final result should look similar to the graph in Figure 5-51.

Figure 5-51. Undo functions and reapply in the correct order

There will be times where you forget to add a function somewhere in the middle of a painfully complex target string. Rather than reconstructing the entire line from scratch, you can instead choose to *Edit* the target manually. Clicking the **Edit** button, or simply double-clicking on a target, will bring up a modal where you can modify the string by hand.

Sharing your work

As you'll see in later chapters, this is a genuinely useful graph for monitoring your Graphite performance. This is the exact same chart that I've referred to time and time again to determine the health of my clusters and debug Carbon or Whisper issues. I urge you to save this graph or, if you're not currently logged in, grab the *Short URL* or the embedded chart link and bookmark them for later use.

Practice what you've learned in this chapter and try to reproduce the results without peeking at these instructions. Get familiar with the functions and how the UI elements map to their render API counterpart. See if you can find different combinations of functions to create an equivalent graph. Practice some more and demonstrate what you've learned for your peers. If you enjoy working with Graphite like I do, there's almost no end to the number of techniques or tips you'll accumulate over time.

CHAPTER 6
Advanced Graph Prototyping

Chapter to Come

CHAPTER 7
Dashboards

Chapter to Come

CHAPTER 8
Whisper Storage

Chapter to Come

CHAPTER 9
Troubleshooting Graphite Performance

People ask me about troubleshooting Graphite performance as if it's some dark art practiced by only the most arcane operators in the bleakest corner of the largest datacenters. Fortunately, nothing could be further from the truth.

Many of Graphite's newer competitors attempt to hide their operational costs with "cluster in a box" designs leaning heavily on NoSQL-type backends. Sadly, many of these projects are immature and have left a wake of data corruption and data loss. In many cases the user is unprepared for the consequences, with little choice but to "blow it all away" and start over again.

By contrast, Graphite embraces traditional UNIX systems to store and retrieve time-series data. It may not be as sexy as the newest datastore on Hacker News, but the use cases, tooling, and failure scenarios are well documented and widely understood. The average systems administrator understands files and filesystems, and how to repair them when something goes sideways. To paraphrase Dan McKinley, *boring technology* just gets the job done.

In this chapter we're going to build a foundation containing the skills and tools you'll need to respond to just about any Graphite troubleshooting or scaling challenge. We'll investigate what happens when you tweak a variety of performance-impacting configuration settings. Every possible log entry has been reproduced and traced back to its source, with examples and explanations provided here for easy reference. Last but not least, we'll look at a variety of failure or degraded situations that you're likely to encounter and discover how to debug them step-by-step.

By the end of this chapter I hope to have illuminated you with years of operational experience running Graphite and Carbon in high-volume production environments. With this foundation built, we can proceed to the next chapter, prepared with the knowledge and experience to design and scale up your own Graphite cluster effectively.

First, the Basics

If you take away nothing else from this chapter, I hope that you remember that troubleshooting Graphite (or more specifically, Carbon) boils down to understanding how and when each of the Carbon process types begins to bottleneck on system resources: disk I/O, memory, and CPU utilization.

The first failure scenario that most new Graphite administrators encounter is when `carbon-cache` is unable to write datapoints to the Whisper files fast enough (or in the case of a full disk, *at all*). What happens when the cache can't flush datapoints in a timely manner? Memory usage balloons until the `MAX_CACHE_SIZE` is met (when set to an integer value), the system runs out of memory (when `MAX_CACHE_SIZE` is set to `inf`), or the cache is able to catch up and write datapoints faster than they're being received. There's a good chance you'll see increased CPU utilization as the cache grows, but this is a secondary concern to your system's ability to handle writes and its memory footprint.

Conversely, the `carbon-relay` and `carbon-aggregator` processes are far more likely to exhibit performance problems due to CPU starvation. They have a queue for each downstream destination, but this is much more conservative than the `carbon-cache`. Datapoints received by a relay or aggregator for a destination where `MAX_QUEUE_SIZE` (number of datapoints) has been reached will either be refused (where `USE_FLOW_CONTROL` is set to `True`) or dropped on the floor. For systems handling many thousands of datapoints per second, the type of processing required can begin taxing the designated CPU core(s).

Most of the confusion I see with people debugging Graphite clusters stems from a lack of understanding of these basic truths, or a lack of experience debugging UNIX systems performance. There's nothing terribly magical or complicated about scaling your Graphite servers once you understand the role and *behavior* of each service. Things can get complicated once we scale out horizontally with HAProxy load balancers, relay replicators, multi-node cache pools, and Graphite render farms, but the underlying personalities of each process never change.

The Troubleshooting Toolbelt

Generating Metrics and Benchmarking

I've stressed earlier in this book that it's vital to know how many metrics your Graphite server is capable of writing. To that end, it's helpful that you understand what your server's disk subsystem can withstand. Without knowing how many I/O operations (commonly referred to as *iops*) your disk can handle, you'll never really know if you've reached saturation of your metrics pipeline. The fact that Carbon can *batch* multiple

datapoints into a `bulk_write` will certainly affect your throughput, but knowing your *practical* limits in terms of iops will eliminate uncertainty.

Most systems administrators I know will reflexively reach for **Bonnie** to benchmark their disk systems. Bonnie is a very handy tool and it's a good idea to run it against your Graphite server, but I find that generating a realistic metrics load is a far more accurate representation of what your Carbon/Whisper service can handle.

For example, running Bonnie against a small-ish DigitalOcean 40GB SSD virtual instance reveals performance between 80-90 iops (according to its *Random Seeks* statistic). A load test of metrics volume using a tool like *Haggar* (discussed below) and monitored with *iostat* shows the same instance is capable of handling upwards of 150 write operations per second. Much of this can be attributed to Bonnie's design; while it's a very useful tool for general I/O benchmarking, its approach varies significantly from the sort of small, ordered writes that Carbon performs.

The **Haggar** project is one of my favorite tools for generating test load of your metrics server, developed by my good friend Michael Gorsuch. Haggar was designed to emulate the ramp-up of a number of collectd "agents" (clients) over a period of time. It allows you to adjust the *flush interval* (how often to submit metrics), number of *metrics* (how many each agent should report), *spawn interval* (delay between new agents), and much more. This gives a very close approximation to what you might see if you were to activate metrics collection for a fleet of servers and is fantastically useful for pinpointing the *point of exhaustion* for any Carbon/Whisper-based cluster.

In the following example, after quickly installing the Go language and Haggar, I run `haggar` against a remote Carbon server. I've intentionally lowered the `-agents` and `-metrics` values to simulate a moderate traffic load. We can see each agent as they launch, and then again as they flush their 200 measurements ten seconds later.

```
$ sudo apt-get install gccgo-go
$ GOPATH=~/gocode go get github.com/gorsuch/haggar
$ ./gocode/bin/haggar -agents=100 -metrics=200 -carbon="10.236.119.223:2003"
2015/05/17 23:58:43 master: pid 15401
2015/05/17 23:58:43 agent 0: launched
2015/05/17 23:58:53 agent 0: flushed 200 metrics
2015/05/17 23:58:55 agent 1: launched
2015/05/17 23:59:03 agent 0: flushed 200 metrics
2015/05/17 23:59:05 agent 1: flushed 200 metrics
2015/05/17 23:59:11 agent 2: launched
2015/05/17 23:59:13 agent 0: flushed 200 metrics
```

This results in a smooth, predictable response from our Carbon service and the system resources it consumes. The `metricsReceived` (number of datapoints received by the `carbon-cache` listener) and `updateOperations` (number of update operations) statistics reflect a linear growth as Haggar brings agents online.

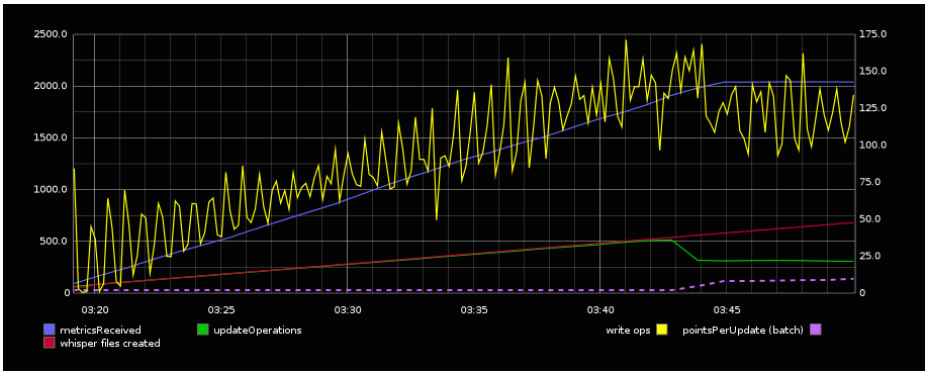


Figure 9-1. Basic haggar usage

However, we begin to see a deviation around `03:43` on the graph. The `updateOperations` begins to trend downward while `pointsPerUpdate` (number of datapoints written per bulk update) increases. In many cases the latter would be a good thing since it signifies that Carbon is trying to be more efficient with its writes. However, the rapid drop in `updateOperations` is a red flag that something else might be at play.

I create a new graph, this time comparing the metric volume with cache and memory usage. Here it becomes obvious that our disk(s) are unable to keep up with the number of write operations requested. The number of cache queues (unique tuple of `carbon-cache` instance and metric names), their aggregate size, and the amount of memory used by the cache all trend upwards.

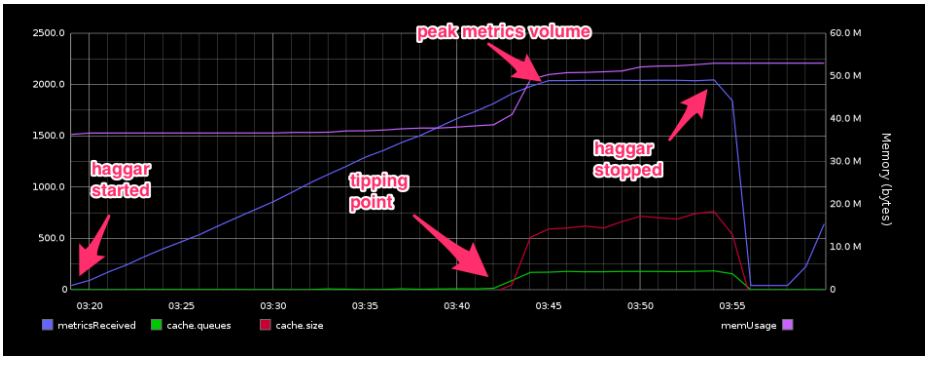


Figure 9-2. Identifying resource exhaustion

Looking deeper we can see exactly what's going on. Shortly after `03:42`, at which point we're seeing approximately 1800 metrics per second, `carbon-cache` is no longer able to send write operations to the disk fast enough (the disk can't keep up). At this point it begins creating cache queues and storing datapoints there. Around `03:45` we reach our peak volume of 2000 metrics per second (100 agents flushing 200 metrics every 10 seconds). This difference of 200 metrics correlates positively with the number of cache queues.

If this Haggar test were allowed to run indefinitely, we could make a pretty fair prediction as to when the system would run out of memory as the cache size continued to grow. Fortunately I was wise enough to kill the test around `03:54`, at which point `carbon-cache` is able to catch up and begin flushing the queues to disk.

Although this was merely a benchmarking exercise, this closely resembles the sort of problems and troubleshooting techniques you should expect to encounter with your own Graphite cluster in a "healthy" production environment. When Graphite becomes popular in your organization, people will want to throw *lots* of metrics at it. This will inevitably lead to capacity and scaling challenges; fortunately for you, this chapter should equip you with the sort of intuitition and Graphite systems expertise normally limited to administrators with years of experience under their belts.

CPU Utilization

The **top** utility is ubiquitous across most UNIX systems as a tool for monitoring system resource utilization on the local system. It's particularly handy for observing CPU and memory consumption in real-time for individual processes like Carbon. The **u** option in a running top session will allow you to filter processes to a specific process user, such as the `carbon` user. You can also toggle the breakout of CPU per-core statistics by entering **1** at any time. This sort of view is something I use consistently to track errant Carbon processes.

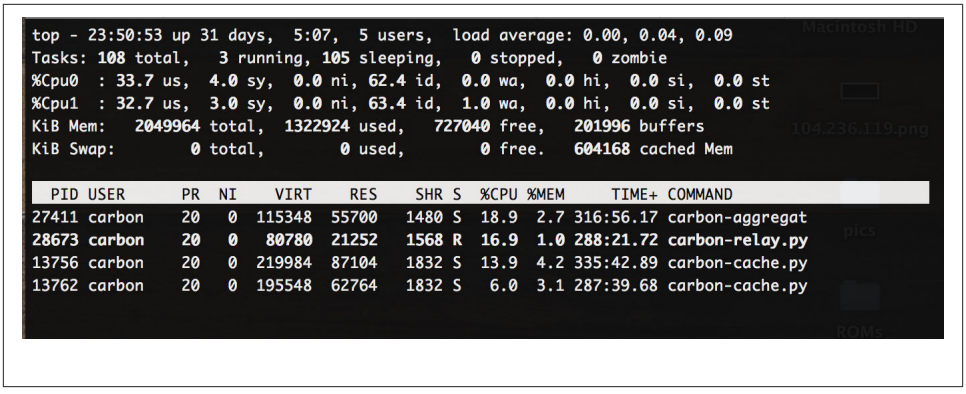


Figure 9-3. Using top to monitor process CPU utilization

Some folks prefer **htop** for its slightly more *visual* approach, such as the horizontal "graph" for per-CPU activity. Either is fine with me although I generally go old school with the original **top**.

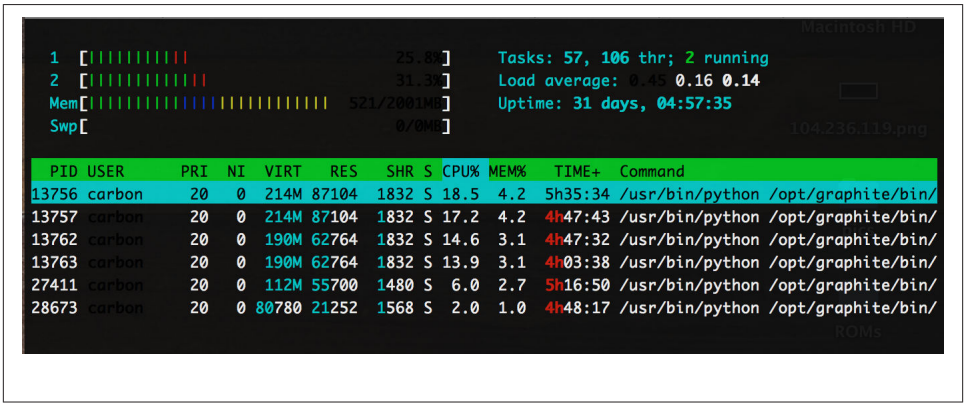


Figure 9-4. The htop utility

Disk Performance (I/O)

As we've seen in the Benchmarking section above, Graphite is very effective at tracking its own I/O statistics. But what happens when Carbon is unable to flush metrics to disk because of a performance bottleneck? System-level tools like **iostat** and **iotop** will always have a place in our toolbox, providing us with real-time introspection of our disk devices and partitions.

iostat

The **iostat** command gathers I/O statistics from the kernel (specifically, from the `/proc` filesystem) which can then be output to the terminal at routine intervals. For example, we can run `iostat -x 1 -y` to view information about all configured disk devices, with a new report generated every second. The `-x` flag displays *extended* statistics, while the `-y` flag instructs iostat to skip the initial summary statistics (since the last boot).

Unless you have a good reason to refresh the output every second, I suggest a longer interval between 5-10 seconds. Shorter intervals will frequently oscillate between zero and peak values, making it difficult to get a good read on how the device is actually performing over time. Summarizing to a longer window gives us a more coherent rolling average that can be useful when trying to determine the practical performance characteristics of the disk system.

If you've not familiar with iostat output, it can be rather confusing. Fortunately there are some very good articles online explaining each of the fields in the utilization report. For our purposes, we're mostly concerned with the following statistics.

%idle
: The percentage of time that CPU(s) were idle and did not have an outstanding disk I/O request.

w/s
: The number of write requests completed per second by the device.

avgqu-sz
: The queue length of requests sent to the device.

w_await
: The average time, in milliseconds, for write requests to be served by the device.

%util
: The percentage of time when I/O requests were issued to the device.

Where `%idle` and `%util` are easy to comprehend and interpret, `w/s`, `avgqu-sz`, and `w_await` require more careful scrutinization. Each of these values are going to be unique to your particular system configuration and environment, so it's impossible to qualify "good" or "bad" values. Rather, you want to monitor them for trends within the context of your overall device performance. Generally speaking, if you see write requests per second plateau, while the queue length and wait time increase, you've probably hit the write ceiling. When this happens it would be a good idea to try and back off some of your metrics sources and see if this allows the system to recover.

The exclusion of *read*-related statistics from the list above isn't meant to imply that these aren't important. They absolutely are, but most of the bottlenecks you're bound to encounter with Carbon will be in the *write path*. Regardless, you should always keep an eye out for any unusual read activity that may be stealing throughput in the I/O pipeline.

The following figure captures iostat output at a point when the `carbon-cache` processes on this system were unable to flush metrics quickly enough, resulting in large cache queues and increased memory footprint. It might be natural to assume that this was due to insufficient disk performance; however, we can clearly see that the CPU core(s) were fully saturated with I/O requests. In this case, the CPU was the performance bottleneck, not the disk devices. The `%util` value backs up this assertion; the disk is effectively sitting around twiddling its thumbs.

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          97.55    0.00    2.30    0.00    0.00    0.15

Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
vda               0.00    51.80    0.00  119.70     0.00   686.00    11.46     0.38    3.15    0.00    3.15   0.06   0.72
```

Figure 9-5. Output from iostat

If this particular system (a DigitalOcean 2-core VPS) had additional cores, we could scale up its I/O capacity easily by increasing the number of `carbon-cache` processes running behind a relay. Unfortunately, in this case, we're already running two caches, a relay, and an aggregator. Our only alternative (besides dialing back our metrics collection) would be to migrate to a more powerful instance with additional cores.

iotop

The **iotop** utility is less comprehensive than iostat, but it's very helpful for measuring I/O bandwidth across all processes, ones owned by a specific user, or individual processes by process id. This makes it an effective tool for monitoring the distribution of work across a pool of `carbon-cache` processes. In the following example, I've run `iotop -u carbon -P` to only show actual processes (ignoring threads) owned by the `carbon` user.

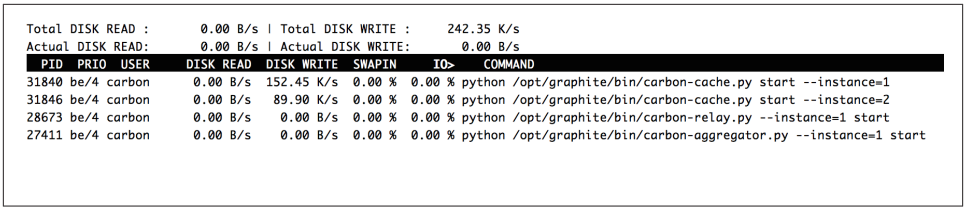


Figure 9-6. Monitoring per-process I/O with iotop

Networking

If you're an old fart like me who worked in the burgeoning ISP industry back in the 1990's, you were almost certainly trained under the Open Sytems Interconnection (OSI) 7-layer model. I rarely hear anyone talk about it outside of networking circles anymore, but it expresses some useful concepts around the abstraction, segmentation, and coordination of responsibilities in *communication systems*.

Each of the seven layers describes a logical separation of functionality. At the lowest level, the *Physical Layer* deals with the transmission and reception of bits over a physical medium. A good example here would be modern Wi-Fi, which takes computer bits and translates them into electrical signals for transmitting through the air.

A little higher up the model, the *Transport Layer* relates to the reliability of a connection, including error control and retransmissions. The most obvious (and successful) example here would be the *Transmission Control Protocol* (TCP), which most of the consumer-facing Internet is built upon.

At the very top of the model, the *Application Layer* includes protocols which support the software and applications directly facing the end user. It does *not* describe the applications themselves, but rather the underlying communications mechanism that allow the applications to perform their networking tasks. Some popular examples here would include HTTP, SMTP, and FTP.

Why do I mention the OSI model at all? Because it impressed upon me the importance of this concept of logical separation, but even more importantly, it taught me to *never assume* that the protocols or services *underneath* were working as intended. This stuck with me, and it reminds me daily to always check the "easy stuff" first before assuming that something more complex (or sinister) was to blame further up the stack.

The next time you're debugging missing metrics between Carbon daemons or from another system or network, make sure to verify that those lower levels are working as expected. There are some basic tools that any modern UNIX/Linux system should have available for troubleshooting.

There are some basic, but surprisingly common, failure scenarios that will catch even the most seasoned professional off-guard if you take your network for granted.

ping

The venerable **ping** utility is common to every networked computer system I'm intimately familiar with. Even small embedded devices come packaged with this tool since it serves a basic, but important purpose in testing network connectivity. Ping works by sending ICMP packets with an *echo request* message and listening for replies with a corresponding *echo response* message.

In the following example we can see a successful ping test. If there were problems related to an unavailable host or network, packet loss, or high latency, we would it reported in the summary statistics.

```
$ ping -c 3 graphite.example.com
PING graphite.example.com (10.131.6.232): 56 data bytes
64 bytes from 10.131.6.232: icmp_seq=0 ttl=41 time=161.189 ms
64 bytes from 10.131.6.232: icmp_seq=1 ttl=41 time=159.376 ms
64 bytes from 10.131.6.232: icmp_seq=2 ttl=41 time=140.744 ms

--- graphite.example.com ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 140.744/153.770/161.189/9.240 ms
```

In some cases you may see misleading results caused by overzealous administrators blocking ICMP traffic on the network or host firewalls. If you think this might be the case, check your firewall rules or reach out to your network administrators for assistance.

One of the side benefits of testing with ping is that it will also report if there are any DNS (Domain Name System) failures, which could cause a problem if you're using hostnames instead of IP addresses in your Graphite or Carbon settings.

```
$ ping -c 3 graphite-bad.example.com
ping: cannot resolve graphite-bad.example.com: Unknown host
```

tcpdump

The **tcpdump** command is easily one of my favorite utilities. Designed to capture network packets as they traverse a network interface, it's useful for inspecting connections to a specific service or port. Because it supports filtering expressions, you can reduce the output to precisely the combination of source and destination attributes you're debugging.

I could go on at length about all the different parameters and options that tcpdump supports, as well as dissecting all of the output columns from the example below. Unfortunately that's a bit out of scope for this book, but I encourage you to search online for any number of "tcpdump primer" articles. At a *very* high level, this output reveals

an established connection between hosts `10.236.102.148` (the client) and `10.236.119.223` (the server listening on port 2003).

```
$ sudo tcpdump -ni eth0 -c 3 port 2003
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes

08:18:12.384631 IP 10.236.102.148.44010 > 10.236.119.223.2003: Flags [P.],
seq 3289726460:3289727836, ack 2169516737, win 115,
options [nop,nop,TS val 1939403604 ecr 163350934], length 1376

08:18:12.384740 IP 10.236.119.223.2003 > 10.236.102.148.44010: Flags [.],
ack 1376, win 726, options [nop,nop,TS val 163353431 ecr 1939403604], length 0

08:18:12.389046 IP 10.236.102.148.44010 > 10.236.119.223.2003: Flags [P.],
seq 1376:2756, ack 1, win 115,
options [nop,nop,TS val 1939403606 ecr 163353431], length 1380
```

Another useful feature of tcpdump is that it will allow you to dump the payload from each packet (the `-X` flag). Every once in a while you might encounter a well-meaning developer, confused as to why the metrics in Graphite don't match up with the way they've instrumented their application. Using tcpdump to capture and display the actual data contents is a quick way to clear up any "confusion".

```
$ sudo tcpdump -nXi eth0 -c 3 port 2003

tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
23:06:02.380915 IP 10.236.102.148.44010 > 10.236.119.223.2003: Flags [P.], seq
223209868:223211183, ack 2169516737, win 115, options [nop,nop,TS val 2514321104
ecr 38268433], length 1315
        0x0000:  4500 0557 ce94 4000 4006 b6c0 68ec 6694  E..W..@.@...h.f.
        0x0010:  68ec 77df abea 07d3 0d4d e98c 8150 32c1  h.w......M...P2.
        0x0020:  8018 0073 2a22 0000 0101 080a 95dd 7ed0  ...s*".........~.
        0x0030:  2c01 1511 636f 6c6c 6563 7464 2e63 6c69  ,...collectd.cli
        0x0040:  656e 742d 302e 7377 6170 2e73 7761 705f  ent-0.swap.swap_
        0x0050:  696f 2d69 6e20 302e 3030 3030 3030 2031  io-in.0.000000.1
        0x0060:  3433 3234 3336 3735 320d 0a63 6f6c 6c65  432436752..colle
        0x0070:  6374 642e 636c 6965 6e74 2d30 2e73 7761  ctd.client-0.swa
        0x0080:  702e 7377 6170 5f69 6f2d 6f75 7420 302e  p.swap_io-out.0.
...
...
```

netstat

The **netstat** command is a jack-of-all-trades for inspecting network interface statistics, routing tables, and more. For our purposes we're mostly interested in its ability to show the state of network sockets: including the IP address(es) they're listening on, how many

established connections they have, and whether there might be a problem with the service itself (e.g. too many connections in the CLOSE_WAIT state).

In the following example we've used netstat to verify that we have services listening on TCP ports 2003 and 2004. The `0.0.0.0` address also tells us that both services are listening on all addresses currently bound to this host.

We also used netstat to view all `ESTABLISHED` connections to port 2003. In this case there are three external clients sending data, in addition to one local client (`127.0.0.1`, also known as the *loopback* interface). Note that this last connection is represented in the netstat output twice, once for the client and another for the server (since they both belong to this host).

```
$ netstat -vant | grep LISTEN | grep ':20'
tcp    0    0 0.0.0.0:2003            0.0.0.0:*               LISTEN
tcp    0    0 0.0.0.0:2004            0.0.0.0:*               LISTEN

$ netstat -vant | grep 2003 | grep ESTAB
tcp    0    0 127.0.0.1:2003          127.0.0.1:42526         ESTABLISHED
tcp    0    0 127.0.0.1:42526         127.0.0.1:2003          ESTABLISHED
tcp    0    0 10.236.119.223:2003     10.131.70.241:44955     ESTABLISHED
tcp    0    0 10.236.119.223:2003     10.236.96.45:35056      ESTABLISHED
tcp    0    0 10.236.119.223:2003     10.236.102.148:44010    ESTABLISHED
```

What about lsof?

Experienced operators might point out that the **lsof** command can return the same information, in addition to the actual name of the process owning the network socket (and tons of other useful stuff). This is certainly true, although I prefer netstat's more concise output for this type of query. If you want to dig deeper for that process name afterwards, lsof is definitely my tool of choice.

```
$ sudo lsof -n -P -i tcp:42526
COMMAND     PID   USER  TYPE NODE NAME
collectd   1109   root  IPv4  TCP 127.0.0.1:42526->127.0.0.1:2003 (ESTABLISHED)
carbon-re 28673 carbon  IPv4  TCP 127.0.0.1:2003->127.0.0.1:42526 (ESTABLISHED)
```

iftop

Similar to the way *iotop* helps us identify I/O bandwidth across a series of processes, **iftop** allows us to do the same for *network* bandwidth across Graphite-Web or Carbon services. This is particularly handy for ensuring an equal distribution of connections across a pool of relay or cache listeners.

```
$ sudo iftop -f "port 2003 or port 2103 or port 2203"
```

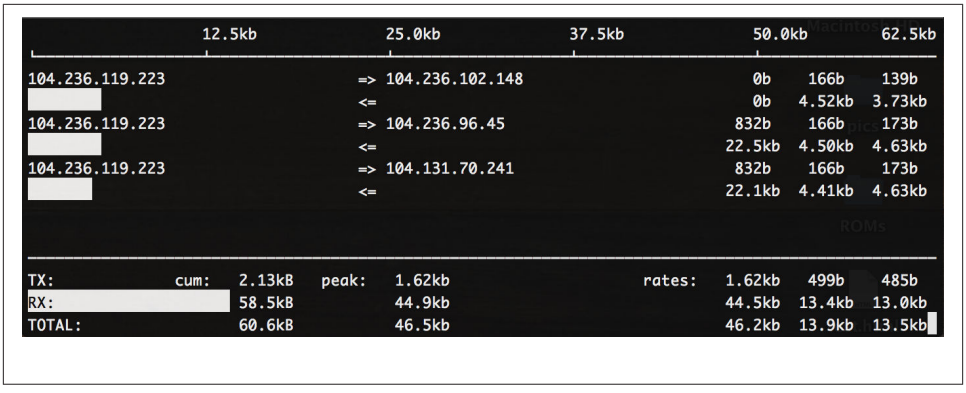


Figure 9-7. Monitoring traffic distribution among carbon-cache deamons

Case in point, say you have a pool of `carbon-relay` listeners behind an HAProxy service. Anytime an aggregator or large metrics source sends a high volume of metrics through a single connection (e.g. statsd or the collectd `network` plugin acting as a central proxy), these metrics will be forced through a single relay. Only after they've been routed to the backend using relay rules or the consistent-hashing algorithm will they achieve any sort of equal distribution. This is a *very* important consideration when designing your metrics pipeline; besides the HA (high availability) implications, simple bottlenecks like these can significantly impact the efficiency of your architecture.

Inspecting Metrics

The Whisper project includes a number of command-line utilities for creating, modifying, and inspecting Whisper database files. There are a few in particular that come in handy for troubleshooting or simply taking a closer look at the data than a rendered graph might permit. Alternatively, if you're a developer that perhaps doesn't have access to the local Whisper filesystem, the Graphite API can be very handy for retrieving the raw data when you need it.

whisper-info.py

Each Whisper file contains metadata about the metric and its respective rollup archives. The **whisper-info.py** script provides easy access to this metadata.

```
$ cd /opt/graphite/storage/whisper/

$ whisper-info.py carbon/agents/synthesize-1/metricsReceived.wsp

maxRetention: 31536000
xFilesFactor: 0.5
aggregationMethod: average
fileSize: 6307228
```

```
Archive 0
retention: 31536000
secondsPerPoint: 60
points: 525600
size: 6307200
offset: 28
```

From this example we can deduce that the `metricsReceived` metric is stored at 1-minute (60 `secondsPerPoint`) resolution for one year. We can see that pretty easily by converting `maxRetention` (reported in number of seconds) into the number of days that Whisper will retain data for this metric.

```
31536000 / 60 (seconds) = 525600 (minutes)
525600 / 60 (minutes) = 8760 (hours)
8760 / 24 (hours) = 365 (days)
```

whisper-fetch.py

If you need to retrieve some metric data, or just want to make sure that measurements are actually being received (and written to disk), **whisper-fetch.py** gets the job done. With no optional arguments, this utility will dump the most recent 24 hours worth of datapoints for the specified metric file. It's important to note that it will pull these metrics from *the first rollup archive can satisfy this range*.

```
$ whisper-fetch.py \
    carbon/agents/synthesize-1/metricsReceived.wsp | tail -5

1432395660      1296.000000
1432395720      1352.000000
1432395780      1316.000000
1432395840      1326.000000
1432395900      1329.000000
```

Fortunately, `whisper-fetch.py` also supports the `--from` and `--until` arguments for defining the range of datapoints to query. These can be used to limit the query range to a desired retention archive.

It also accepts arguments such as `--pretty` and `--json` to adjust the formatted output as required.

```
$ whisper-fetch.py --from=1432395540 --pretty \
    carbon/agents/synthesize-1/metricsReceived.wsp

Sat May 23 11:40:00 2015        1338.000000
Sat May 23 11:41:00 2015        1296.000000
Sat May 23 11:42:00 2015        1352.000000
Sat May 23 11:43:00 2015        1316.000000
```

```
Sat May 23 11:44:00 2015        1326.000000
Sat May 23 11:45:00 2015        1329.000000

$ whisper-fetch.py --from=1432395540 --json \
    carbon/agents/synthesize-1/metricsReceived.wsp

{
    "start" : 1432395600,
    "end" : 1432396200,
    "step" : 60,
    "values" : [1338.0, 1296.0, 1352.0, 1316.0, 1326.0, 1329.0]
  }
```

whisper-dump.py

If you prefer to view *everything* at once, **whisper-dump.py** is your tool of choice. It reports the same metadata information as `whisper-info.py`, but then it also dumps all data from each archive. It accepts no optional arguments, so what you see is what you get.

```
$ whisper-dump.py collectd/client-0/memory/memory-used.wsp | less

Meta data:
  aggregation method: average
  max retention: 31536000
  xFilesFactor: 0.5

Archive 0 info:
  offset: 40
  seconds per point: 10
  points: 60480
  retention: 604800
  size: 725760

Archive 1 info:
  offset: 725800
  seconds per point: 60
  points: 525600
  retention: 31536000
  size: 6307200

Archive 0 data:
0: 1432271980,  117186560
1: 1432271990,  117264384
2: 1432272000,  116436992
...

Archive 1 data:
0: 1420176000,   69343232
1: 1420176060,   69341184
```

```
2: 1420176120,   69378048
...
```

API text formats

For the user who has everything (except root access to their servers), the Graphite API offers a variety of text outputs using the **format** parameter. The **raw** output is designed for efficiency, conveying the query's metric target, the start and stop times, the precision interval, and a list of all datapoint values, in a single line. The **csv** output is straightforward, and can be used to import directly into CSV-compatible applications. Last but not least, the **json** target is particularly popular for enabling client-side rendering with D3.js and third party dashboards.

```
$ curl "http://127.0.0.1/render/?from=-300s&target=test.foo&format=raw"
test.foo.bar,1432396560,1432396860,60|1325.0,1309.0,1315.0,1362.0,1312.0

$ curl "http://127.0.0.1/render/?from=-300s&target=test.foo&format=csv"
test.foo,2015-05-23 16:00:00,1312.0
test.foo,2015-05-23 16:01:00,1342.0
test.foo,2015-05-23 16:02:00,1314.0
test.foo,2015-05-23 16:03:00,1328.0
test.foo,2015-05-23 16:04:00,1333.0

$ curl -s "http://127.0.0.1/render/?from=-300s&target=test.foo&format=json"
[
   {
      "datapoints" : [
         [
            1362,
            1432396740
         ],
         [
            1312,
            1432396800
         ],
         [
            1342,
            1432396860
         ],
         [
            1314,
            1432396920
         ],
         [
            1328,
            1432396980
         ]
      ],
      "target" : "test.foo"
```

```
      }
    ]
```

Configuration Settings

Carbon

```
LOG_UPDATES
LOG_CACHE_HITS
LOG_CACHE_QUEUE_SORTS
LOG_LISTENER_CONNECTIONS
MAX_CACHE_SIZE
MAX_UPDATES_PER_SECOND
MAX_CREATES_PER_MINUTE
WHISPER_AUTOFLUSH
WHISPER_SPARSE_CREATE
WHISPER_FALLOCATE_CREATE
WHISPER_LOCK_WRITES
MAX_DATAPOINTS_PER_MESSAGE
MAX_QUEUE_SIZE
QUEUE_LOW_WATERMARK_PCT
USE_FLOW_CONTROL
CACHE_WRITE_STRATEGY
    ==== Graphite-Web
LOG_CACHE_PERFORMANCE
LOG_RENDERING_PERFORMANCE
LOG_METRIC_ACCESS
    CARBONLINK_QUERY_BULK
```

Internal Carbon Statistics

carbon-cache

carbon-relay

carbon-aggregator

Graphite Statistics

logging

kernel

```
Apr 13 18:37:24 synthesize kernel: [6585539.653303] Out of memory: Kill process 1262 (carbon-cache
Apr 13 18:37:24 synthesize kernel: [6585539.653403] Killed process 1262 (carbon-cache.py) total-vm
Apr 21 00:51:35 synthesize kernel: [108516.966575] TCP: TCP: Possible SYN flooding on port 2003. S
```

carbon-cache

console.log

```
13/04/2015 23:27:26 :: Log opened.
13/04/2015 23:27:26 :: twistd 11.0.0 (/usr/bin/python 2.7.6) starting up.
13/04/2015 23:27:26 :: reactor class: twisted.internet.epollreactor.EPollReactor.
13/04/2015 23:27:26 :: twisted.internet.protocol.ServerFactory starting on 2003
13/04/2015 23:27:26 :: Starting factory <twisted.internet.protocol.ServerFactory instance at 0x7ff
13/04/2015 23:27:26 :: twisted.internet.protocol.ServerFactory starting on 2004
13/04/2015 23:27:26 :: Starting factory <twisted.internet.protocol.ServerFactory instance at 0x7ff
13/04/2015 23:27:26 :: twisted.internet.protocol.ServerFactory starting on 7002
13/04/2015 23:27:26 :: Starting factory <twisted.internet.protocol.ServerFactory instance at 0x7ff
13/04/2015 23:27:26 :: set uid/gid 998/998


14/04/2015 00:01:05 :: 'Error creating /opt/graphite/storage/whisper/haggar/agent/15/metrics/57.ws
14/04/2015 00:01:05 :: close failed in file object destructor:
14/04/2015 00:01:05 :: IOError: [Errno 28] No space left on device


19/04/2015 23:39:50 :: MetricCache is full: self.size=1223


19/04/2015 22:56:49 :: Sorted 87 cache queues in 0.000146 seconds


21/04/2015 00:31:48 :: 'Failed to parse line: foo'


21/04/2015 01:00:23 :: /opt/graphite/conf/storage-aggregation.conf not found, ignoring.


21/04/2015 01:07:59 :: Invalid schemas found in foobar: Higher precision archives' precision must


21/04/2015 01:15:07 :: No storage schema matched the metric 'foobar.test', check your storage-sche


21/04/2015 01:52:33 :: Could not accept new connection (EMFILE)
```

creates.log

```
14/04/2015 00:00:02 :: new metric haggar.agent.25.metrics.48 matched schema default
14/04/2015 00:00:02 :: new metric haggar.agent.25.metrics.48 matched aggregation schema default_av
14/04/2015 00:00:02 :: creating database file /opt/graphite/storage/whisper/haggar/agent/25/metric
```

listener.log

```
19/04/2015 00:00:07 :: MetricLineReceiver connection with 127.0.0.1:58429 established
19/04/2015 00:00:07 :: MetricLineReceiver connection with 127.0.0.1:58429 closed cleanly


19/04/2015 16:51:46 :: MetricLineReceiver connection with 10.131.70.241:46278 lost: Connection to
19/04/2015 16:51:46 :: MetricLineReceiver connection with 10.236.102.148:38410 lost: Connection to
19/04/2015 16:51:46 :: MetricLineReceiver connection with 127.0.0.1:40860 lost: Connection to the
19/04/2015 16:51:46 :: MetricLineReceiver connection with 10.236.96.45:53909 lost: Connection to t


21/04/2015 00:54:18 :: invalid line received from client 127.0.0.1:37044, ignoring
```

query.log

```
19/04/2015 06:39:02 :: 127.0.0.1:52946 disconnected
19/04/2015 16:49:28 :: 127.0.0.1:60245 connected


19/04/2015 16:51:46 :: 127.0.0.1:60245 connection lost: Connection to the other side was lost in a


19/04/2015 16:53:37 :: [127.0.0.1:60385] cache query for "carbon.agents.synthesize-1.metricsReceiv
19/04/2015 23:00:30 :: [127.0.0.1:39080] cache query for "carbon.agents.synthesize-1.cache.queues"
```

updates.log

```
19/04/2015 23:55:33 :: wrote 1 datapoints for collectd.client-0.cpu-0.cpu-wait in 0.00009 seconds
```

carbon-relay and carbon-aggregator

clients.log

```
20/04/2015 23:37:22 :: CarbonClientFactory(127.0.0.1:2024:1)::startedConnecting (127.0.0.1:2024)
20/04/2015 23:37:22 :: CarbonClientProtocol(127.0.0.1:2024:1)::connectionMade


20/04/2015 23:37:15 :: CarbonClientFactory(127.0.0.1:2104:1)::startedConnecting (127.0.0.1:2104)
20/04/2015 23:37:15 :: CarbonClientFactory(127.0.0.1:2204:2)::startedConnecting (127.0.0.1:2204)
20/04/2015 23:37:15 :: CarbonClientProtocol(127.0.0.1:2104:1)::connectionMade
20/04/2015 23:37:15 :: CarbonClientProtocol(127.0.0.1:2204:2)::connectionMade
```

```
21/04/2015 00:43:36 :: CarbonClientFactory(127.0.0.1:2204:2) send queue is full (10000 datapoints)

21/04/2015 00:44:37 :: CarbonClientProtocol(127.0.0.1:2204:2) send queue has space available
```

console.log

listener.log

aggregator.log

```
21/04/2015 00:03:40 :: Allocating new metric buffer for collectd.client-0.cpu.all.cpu-user

21/04/2015 00:00:45 :: reading new aggregation rules from /opt/graphite/conf/aggregation-rules.con
21/04/2015 00:00:45 :: clearing aggregation buffers
```

webapp

cache.log

```
Tue Apr 21 15:19:07 2015 :: CarbonLink creating a new socket for ('127.0.0.1', None)

Mon Apr 20 03:00:30 2015 :: CarbonLink cache-query request for carbon.agents.synthesize-1.cache.qu

Tue Apr 21 15:19:18 2015 :: Request-Cache hit [99430ed53968dacab9f6a18e5e76b49e]
Tue Apr 21 15:20:18 2015 :: Request-Cache miss [99430ed53968dacab9f6a18e5e76b49e]

Tue Apr 21 15:20:18 2015 :: Data-Cache miss [adea338bbe2f6cc11f35cb3da201c1b7]
Tue Apr 21 16:05:48 2015 :: Data-Cache hit [769db380c1d2622855cc7ab49bdf0432]
```

exception.log

```
Mon Apr 20 00:20:49 2015 :: Failed CarbonLink query 'carbon.agents.synthesize-1.creates'
Traceback (most recent call last):
  File "/opt/graphite/webapp/graphite/render/datalib.py", line 379, in fetchData
    cachedResults = CarbonLink.query(dbFile.real_metric)
  File "/opt/graphite/webapp/graphite/render/datalib.py", line 144, in query
    results = self.send_request(request)
  File "/opt/graphite/webapp/graphite/render/datalib.py", line 206, in send_request
    result = self.recv_response(conn)
  File "/opt/graphite/webapp/graphite/render/datalib.py", line 218, in recv_response
    len_prefix = recv_exactly(conn, 4)
  File "/opt/graphite/webapp/graphite/render/datalib.py", line 233, in recv_exactly
    raise Exception("Connection lost")
Exception: Connection lost
```

info.log

```
Mon Apr 20 00:45:31 2015 :: [IndexSearcher] performing initial index load
Mon Apr 20 00:45:31 2015 :: [IndexSearcher] reading index data from /opt/graphite/storage/index
Mon Apr 20 00:45:31 2015 :: [IndexSearcher] index reload took 0.001499 seconds (293 entries)


Tue Apr 21 15:11:59 2015 :: find_view query=* local_only=0 matches=4
Tue Apr 21 15:12:01 2015 :: find_view query=collectd.* local_only=0 matches=4
Tue Apr 21 15:12:03 2015 :: find_view query=collectd.graphite.* local_only=0 matches=16
Tue Apr 21 15:12:05 2015 :: find_view query=collectd.graphite.memory.* local_only=0 matches=4


Wed Apr 22 04:03:32 2015 :: myGraphLookup: username=admin, path=, userpath_prefix=, 1 graph to pro
Wed Apr 22 04:03:43 2015 :: myGraphLookup: username=admin, path=test., userpath_prefix=test., 1 gr
```

rendering.log

```
Wed Apr 22 04:03:17 2015 :: Retrieval of collectd.client-0.cpu.all.cpu-* took 0.065637
Wed Apr 22 04:03:18 2015 :: Rendered PNG in 0.337961 seconds
Wed Apr 22 04:03:18 2015 :: Total rendering time 0.474211 seconds


Wed Apr 22 04:03:44 2015 :: Returned cached response in 0.005794


Wed Apr 22 05:07:40 2015 :: Total rawData rendering time 0.046805


Wed Apr 22 05:09:07 2015 :: Total pickle rendering time 0.164042


Thu Apr 23 02:12:58 2015 :: Retrieval of collectd.client-0.cpu.all.cpu-user took 0.767433
Thu Apr 23 02:12:58 2015 :: Retrieval of collectd.client-0.cpu.all.cpu-softirq took 0.115058
Thu Apr 23 02:12:59 2015 :: Remotely rendered image on 10.236.119.223:80 in 0.371119 seconds
Thu Apr 23 02:12:59 2015 :: Spent a total of 0.378260 seconds doing remote rendering work
Thu Apr 23 02:12:59 2015 :: Total rendering time 1.273846 seconds


Thu Apr 23 02:12:58 2015 :: Retrieval of collectd.client-0.cpu.all.cpu-user took 0.028127
Thu Apr 23 02:12:58 2015 :: Total pickle rendering time 0.049458
Thu Apr 23 02:12:58 2015 :: Retrieval of collectd.client-0.cpu.all.cpu-softirq took 0.019578
Thu Apr 23 02:12:58 2015 :: Total pickle rendering time 0.030675
Thu Apr 23 02:12:59 2015 :: Rendered PNG in 0.346798 seconds
Thu Apr 23 02:12:59 2015 :: Delegated rendering request took 0.363177 seconds
```

Failure Scenarios

The Full Disk

```
Traceback (most recent call last):
  File "/usr/local/lib/python2.7/dist-packages/django/core/handlers/base.py", line 111, in get_response
    response = callback(request, *callback_args, **callback_kwargs)
  File "/opt/graphite/webapp/graphite/render/views.py", line 122, in renderView
    seriesList = evaluateTarget(requestContext, target)
  File "/opt/graphite/webapp/graphite/render/evaluator.py", line 10, in evaluateTarget
    result = evaluateTokens(requestContext, tokens)
  File "/opt/graphite/webapp/graphite/render/evaluator.py", line 21, in evaluateTokens
    return evaluateTokens(requestContext, tokens.expression)
  File "/opt/graphite/webapp/graphite/render/evaluator.py", line 24, in evaluateTokens
    return fetchData(requestContext, tokens.pathExpression)
  File "/opt/graphite/webapp/graphite/render/datalib.py", line 372, in fetchData
    dbResults = dbFile.fetch(startTime, endTime, now)
  File "/opt/graphite/webapp/graphite/storage.py", line 331, in fetch
    return whisper.fetch(self.fs_path, startTime, endTime, now)
  File "/usr/local/lib/python2.7/dist-packages/whisper.py", line 715, in fetch
    return file_fetch(fh, fromTime, untilTime, now)
  File "/usr/local/lib/python2.7/dist-packages/whisper.py", line 719, in file_fetch
    header = __readHeader(fh)
  File "/usr/local/lib/python2.7/dist-packages/whisper.py", line 220, in __readHeader
    raise CorruptWhisperFile("Unable to read header", fh.name)
CorruptWhisperFile: Unable to read header (/opt/graphite/storage/whisper/haggar/agent/20/metrics/406.wsp)
```

Figure 9-8. Backtrace for corrupted Whisper file

```
$ cd /opt/graphite/storage/whisper/

$ whisper-info.py ./haggar/agent/20/metrics/406.wsp
[ERROR] Unable to read header (./haggar/agent/20/metrics/406.wsp)

$ ls -l ./haggar/agent/20/metrics/406.wsp
-rw-r--r-- 1 carbon carbon 0 May 23 09:54 ./haggar/agent/20/metrics/406.wsp
```

CPU Core Saturation

Running out of Files or Inodes

Gaps in Graphs

Rendering Problems

CHAPTER 10
Scaling Graphite

Chapter to Come

CHAPTER 11

Appendix A - The Render API

Appendix to Come

CHAPTER 12

Appending B - Telemetry Toolbox

Appendix to Come

CHAPTER 13

Appending B - Telemetry Toolbox

Appendix to Come