O'REILLY®

Lightweight Systems for Realtime Monitoring

Sam Newman

Lightweight Systems for Realtime Monitoring

Sam Newman

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

Lightweight Systems for Realtime Monitoring

by Sam Newman

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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides

May 2014: First Edition

Revision History for the First Edition:

2014-05-26: First release

ISBN: 978-1-491-94529-2

[LSI]

Table of Contents

Lightweight Systems for Realtime Monitoring

We are surrounded by data. It's everywhere. In our browsers, our databases, lying around on our machines in the form of logs. It sits in memory on application servers and flows across our organizations through emails and is trapped in log files. Individually, that data only has value when it can be accessed, analyzed, and understood. Different silos of data all have different mechanisms by which to read and process them. From the human eye to SQL queries or Hadoop jobs, we've gotten better at processing this data, even at scale. But all too often, this data still lives and is processed in its silos.

The next level of understanding comes from breaking down the barriers that surround our data, making it more open and accessible—this allows us to map one data set against another, to look for correlation that can hopefully lead to an understating of causation and a greater awareness of what's happening. The challenge is the effort required to free the data. We're using the same old siloed mindset when we think about the tools being used and how people will want to access the data.

This paper discusses an approach to making access and understanding of the data we already have more immediate and more valuable. It looks at existing tools and use cases and attempts to point in a direction where things are already headed. It imagines a world where data isn't locked up in secure locations with tool-specific interfaces, but where instead our data flows freely across our networks as events, routed over more generic simple protocols, with a whole suite of multi-purpose tools that can be used to analyze and derive understanding.

Operations and Business—One World Divided

The data silos mentioned previously are rarely more evident than when we consider the separation that occurs between the traditional analytics and data warehouse teams and the world of IT operations. The former plays a business-facing role, hoping to provide insight and intelligence to allow organizations to understand not only how their organizations are performing, but also to help them decide where to go next.

> "We're seeing increased traffic from China—perhaps we should build a Chinese-language website?"
>
> "No one is clicking through to our video page—do we need to redesign our navigation?"
>
> "Shipment times to Ohio are way up! We need more trucks!"

Small, focused Open Source tools and frameworks are charting the direction to a new, more accessible approach to deriving understanding from our systems. No longer the purview of specialist teams using expensive software and hardware, the democratization of analytics is well under way.

To understand what's possible, it's important to take a look at the tools being used in this space. We'll be looking into some broad categories of tooling to do this, including Trending, Dashboards, Event Aggregation, and the emerging space of Anomaly Tracking. These are all open source tools that have emerged from the needs of Operation teams but that are finding increasing use in understanding our business systems.

Graphite

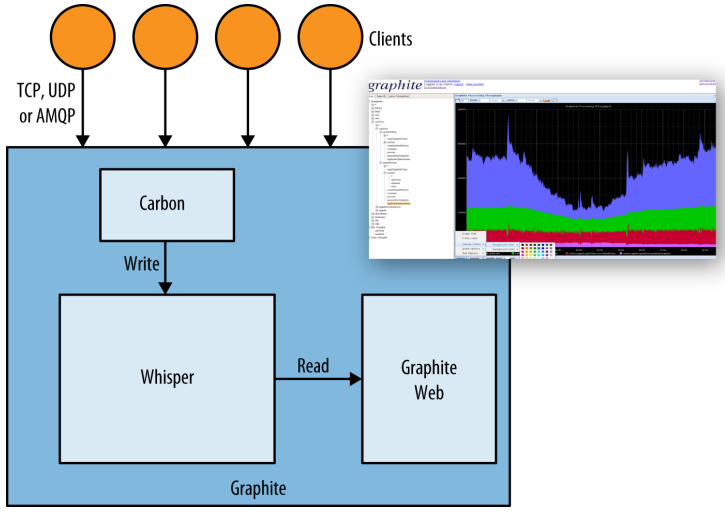
Graphite is a widely used, near real time metrics-gathering system. Written in Python, it operates in a similar space to tools like Munin or Ganglia, allowing metrics to be captured from multiple machines, giving you the ability to drive down to a single source of data, or else take a high-level view.

Although typically used to capture information like CPU or memory use, Graphite is completely agnostic about the nature of the data being stored in it. Its flexibility is partly a result of its incredibly simple data

schema. Each value in Graphite consists simply of a metric name, a value, and a timestamp. By convention, the metric name is delimited into a folder-like structure. For example:

```
mymachine.user_cpu 45 1286269200
```

Graphite itself consists of three parts. Carbon is the daemon process that receives the metrics. These are then stored in Whisper, which is the backing store. Finally, the Graphite Dashboard allows you to visualize the data and create queries. All these parts of the system can be scaled independently, allowing for large volumes of data to be collected in near real time.



The Whisper aggregating backend is particularly interesting. It's based on some of the same principles used in round robin databases (like RRDTool). The idea behind Whisper is to allow you to see metrics from a long time ago, without having to constantly add new storage. Whisper allows you to specify retention times for your metrics, specifying when and how to aggregate up old values to keep space increase to a minimum. For example, you might want one CPU sample every second for the last day, one sample every minute for the last month, but only one every 30 minutes for the last 2 or 3 years. So when data is most timely, where having fine-grained data is most important, you can get at that data. But for older records, where the overall trending is more important than a high degree of fidelity, well you can keep that around without having huge storage requirements.

The Graphite dashboard has some nice tricks up its sleeve. It lets you explore the available metrics, performing various functions on the data. The resulting line graphs are then served up as images that can be bookmarked; reloading the image gets you the new data, making it easy to embed Graphite Dashboard graphs in existing pages or dashboards.

Easy In, Easy Out

One of the reasons why Graphite has been so successful is that its schema for storing metrics is so simple, and adding data from new sources doesn't require any changes on the server. Simply open up a TCP or UDP connection and send the data in. This is especially attractive in an environment where you are provisioning nodes in a dynamic fashion. Graphite's simple data capture schema has led to a number of supporting tools. Notable examples include:

NSClient++
: This is the Windows Daemon for Nagios clients, which supports Graphite as a destination for operating system metrics.

collectd
: The standard metrics collection tool for Linux operating systems has supported Graphite for a number of years via third-party plugins, and since version 5, support for Graphite comes with the standard install.

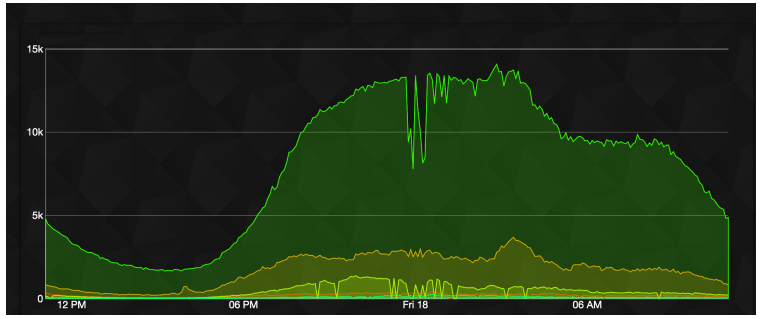
Yammer's Metrics Library
: This is a Java library for collecting in-process metrics, a technique we'll talk more about later. It supports Graphite as a destination for these metrics.

Logstash
: Logstash can parse metrics out of log files and send them to Graphite.

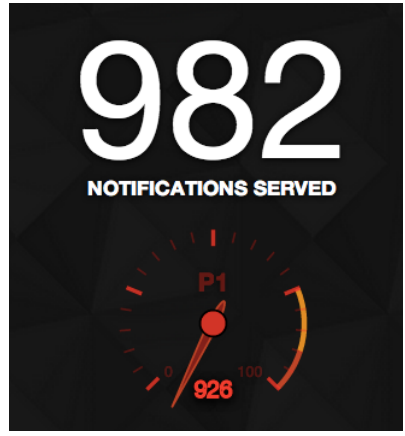
Graphite's own graphs, though, aren't terribly pretty, and for more realtime data, people like to see a more active graph. Also, you only get line graphs out of the tool. Luckily, as with the simple input schema, it's a simple job to get raw data out of Graphite. If you want the raw data as JSON, for example, this is supported out of the box, making creating your own displays much easier.

This has allowed many other people to create separate dashboard and graphing tools to create more interesting dashboards on top of Graph-

ite. Graphene is a D3-based static site that displays moving line charts of Graphite data:



Graphene is also capable of displaying Graphite data as simple numbers, or even gauges. These examples come from the standard demo:



This is highly useful—sometimes a line chart makes sense, while other times a big number is what you want.

Graphite's success has lain in being very focused in what it does and providing a simple schema for collecting and sharing data.

LogStash

For some of us, our application log files are a graveyard of cruft, littered with the hangover of development, errors that no one ever looks at, and the occasional *Hi, Mom!* log message. They can be a hugely valuable resource, however. Apache log files, for example, can show you response codes and response time for calls made—vital for under-

standing if a system is behaving well. Well-maintained log statements in our own applications can be similarly useful.

One of the core challenges, though, is that logs are too often used in a passive way; they're used when a problem has already been identified elsewhere. The log files are not in our eye line the same way dashboards are—they're *over there*, on the machines themselves. I have actually seen log files referred to as a problem more than a source of valuable information ("they just keep growing!").

At scale, even if you just want to use your logs for after-the-fact problem identification, that can become a problem. Logging on to one or two boxes to get the log files isn't too bad, but what about if you had 10, 20, or over 100 machines to get log files from?

LogStash is one of a number of tools that allow you to collect and aggregate log files to a central location to make analysis easier. When combined with querying tools like Kibana or GrayLog2, you can end up with a highly queryable frontend to your logs. In this way, LogStash and other tools play in the same space as the very good (albeit the often very expensive commercial tool) Splunk.

LogStash works based on input, output, and filter plugins. Input plugins allow you to get the data in the first place: from a file, a TCP socket, or stdin. Filters process and change the logs they're sent, allowing you to create more queryable data. The Grok filter, for example, lets you extract bits of data from unstructured log lines, ignoring the junk information and giving a more structured, information-rich result. Finally, the output plugins allow you to specify where your data gets sent to, which includes databases, alerting systems, or even email. It could just consolidate everything into another file, send it into an elasticsearch instance to allow for rich querying, or forward to another system for more processing.

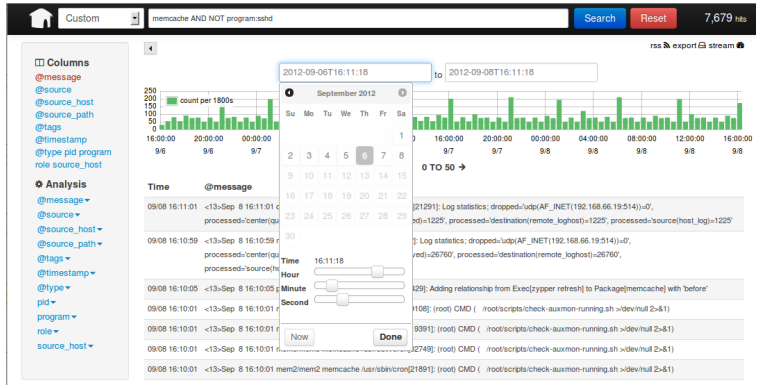
Some output plugins let you send information to other commonly used systems that aren't typically associated with logs. For example, the Nagios output plugin lets you infer the health of a system from logs and tell Nagios about it so it can alert if needed. The Grapite output lets you send metrics parsed out of logs for storage in Graphite. This flexibility in destination systems allows you to move logs from a passive, after-the-fact tool to something that becomes an active part of your system. All of a sudden you're able to react because of something in your logs. Increasing response times? Perhaps that's an actionable

issue. What about a sudden increase in users clicking on the Support page?

Due to Logstash's highly flexible—albeit very simple—architecture, it can extend out from the space of *log aggregation*. For example, the Twitter input plugin lets you parse tweets from Twitter's streaming API. This could be an important part of an active monitoring system, reporting incidents of how many times your company name is mentioned on Twitter. If it spikes, there could be a problem!

This idea of gathering data from multiple sources, filtering and aggregating it, and forwarding it on is an important one, and one we'll come back to later.

LogStash itself is just a collecting, filtering, forwarding daemon; without something with which to view the collected data, it is of limited use. As with Graphite, an ecosystem of tools that can work with LogStash has emerged. (Or more correctly, LogStash has implemented support for a number of different query/viewer tools.) Historically GrayLog2 was used heavily with LogStash for this purpose, but more recently Kibana, an ElasticSearch-backed backend has emerged as the tool of choice when using LogStash.



StatsD

Graphite's extremely simple featureset has been one of the main reasons for its success. However, its focus on supporting operational metrics does limit its usefulness in other situations. Graphite at its heart relies on preaggregated metrics. For example, when monitoring CPU rates from a machine, the host doesn't send you a new measure

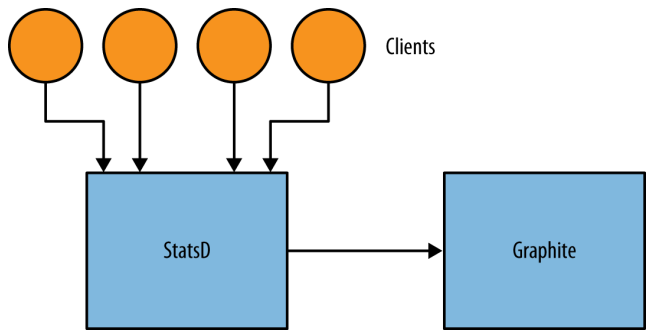
every time the CPU usage changes, but it will typically send you an average every few seconds.

If you send multiple values for the same metric at the same time, Graphite ignores all but the last one it receives. For example, let's imagine we want to record the fact that an order was placed. We might send something like this:

```
orderplaced 1 1286269200
```

Now, if another order is placed within a second (something that seems possible in a moderately sized systems) and we send another `orderplaced 1` value to Graphite for the same timestamp, Graphite will just assume the last value is the right one and won't actually aggregate the values. Net result: two orders were placed and two events sent to Graphite, but Graphite shows only one order being placed.

StatsD, developed by Etsy, is a Node.js port of an earlier Perl tool; it acts as a proxy for Graphite. Its use of Node.js—an evented IO server—allows it to handle potentially thousands of concurrent requests. It is designed to act as a proxying aggregation server—rather than sending metrics to Graphite, you instead send them to StatsD, which does the aggregation for you.



Like Graphite, it has a simple (albeit different) schema. It does away with the need to send a timestamp; instead, you specify the type of metric you're storing. For our `ordersplaced` example, StatsD supports *counters*. To increment the `ordersplaced` metric for the given point of time, you can send the following via X or Y:

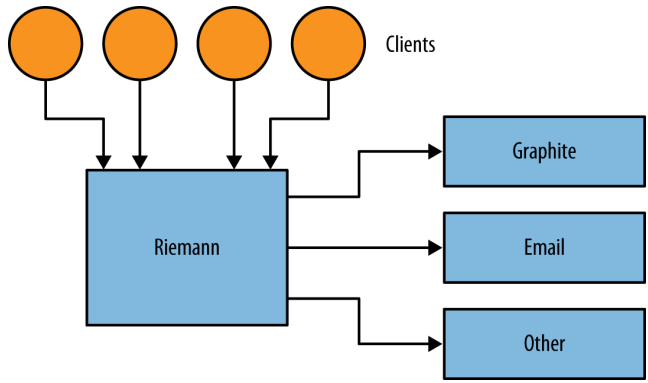
```
ordersplaced:1|c
```

The `c` tells StatsD to consider `ordersplaced` as a counter. StatsD will increment the value it holds for `ordersplaced` before flushing it through to Graphite. In addition to counters, other StatsD types include *gauges* and *timings*. Gauges allow you to send arbitrary values, which will continuously be flushed, until you send a new value. This is useful when you may only be able to sample the source more sporadically than you flush to Graphite. By sending timing metrics, StatsD will automatically generate average, mean, standard deviation, and various other percentiles. This is highly useful when generating things like performance histograms.

Initially, StatsD was built just for Graphite, but it now supports multiple backends via third-party extensions. Supported backends include Mongo, Leftronic, and Ganglia. It also supports sending information to other StatsD nodes, allowing you to run chains of StatsD servers; this makes it possible to handle huge loads of realtime metrics. StatsD, like logstsash, is playing the role of filtering and aggregation system, albeit with some distinct differences.

Riemann

If we were to use the analogy of knives to describe these tools, where StatsD was a single-bladed pocket knife, Riemann is a Swiss Army knife that Macguver would be proud of. On the face of it, it shares a lot in common with StatsD—it is an aggregating, relaying server that can sit in front of Graphite. Like StatsD, it's based on an evented IO model, allowing it to potentially handle thousands of concurrent connections on a single instance. Where the differences come in are the protocol used to talk to Riemann, the way it is configured, and the things you can do with the events it receives.



First, Riemann eschews StatsD and Graphite's simple text-based protocol in favor of a protocol buffer payload that can be sent over either TCP or UDP. This payload is also more complex, containing additional information:

Events

Events are just structs. They're sent over Protocol Buffers, and in Riemann are treated as immutable maps. Each event has these (optional) fields:

host	A hostname, e.g. "api1", "foo.com"
service	e.g. "API port 8000 reqs/sec"
state	Any string less than 255 bytes, e.g. "ok", "warning", "critical"
time	The time of the event, in unix epoch seconds
description	Freeform text
tags	Freeform list of strings, e.g. ["rate", "fooproduct", "transient"]
metric	A number associated with this event, e.g. the number of reqs/sec.
ttl	A floating-point time, in seconds, that this event is considered valid for. Expired states may be removed from the index.

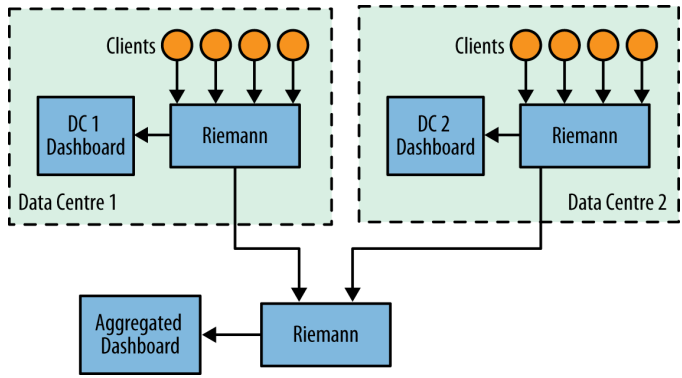
Protocol buggers are a binary-serialization protocol known for compact payloads and the ability to handle versioning in a fairly resilient fashion. They do rely on both the sender and receiver to know the schema of the payload, which makes creating new consumers more complex than the simple text-based protocols. The additional advantage, however, is that they offer more information in terms of structure and type—you can nest information inside a Proto Buffer, allowing (in Riemann's case) a list of tags about the metric. This allows consumers to send far more rich information to Riemann, which Riemann can in turn use to make decisions about how to process the events it receives.

The second key way in which Riemann differentiates itself from StatsD is the way in which events can be processed. Events are pattern matched and processed using functions defined in the Clojure language. These rules can be changed at runtime, and you have the full power of the Clojure general purpose programming language at your disposal.

These Clojure functions can be used to pattern match on the events received, and they perform virtually any action. Riemann can send alert emails when thresholds are reached, generate percentile timings, aggregate and forward data to Graphite, or even forward data to other Riemann nodes. It is highly extensible, and you can call pretty much

any code you can run on the JVM to handle your events. This even opens up the possibility of embedding custom code specific to your problem domain inside Riemann itself.

The option of chaining Riemann nodes together could be a good approach to collecting a large number of events, or it might be appropriate in a situation where you want to aggregate events from multiple different shards or data centers. Forwarding data from multiple Riemann nodes to a single node means you can process an amount of data that might overload a single Riemann instance, as each node in the chain does some of the processing before the final node gets to aggregate all the data together.



These are examples of how powerful Riemann can be in event handling. In this way, Riemann is more akin to a class of systems that provide what is typically termed *complex event processing*. What Riemann provides, though, is built-in support for managing operational and business metrics. Support is also being added to expose data from existing tools, including:

- Cassandra
- HBase
- JMX
- Puppet
- Logstash
- MySQL

While Riemann can be used in a very simple way, it clearly has more potential to become something more in a given setup than StatsD. It

remains to be seen, however, whether the more complex binary protocol it uses will end up being a help or a hindrance in terms of interoperability, which we have seen is key. It is also fair to say that the use of Clojure will be a barrier to some, although the flexibility and power of Riemann comes directly from the capabilities that Clojure itself provides.

One key thing to consider here is the capability for you to model and process events specific to your domain. Riemann gives you a platform to process and derive understanding from the data it receives like nothing else featured in this paper.

Resiliency

Both StatsD and Riemann currently suffer from one drawback when considering the needs for a fully resilient system. By their very nature, these aggregating relays are stateful—they have to store state in order to aggregate events together. Both of them store this state in memory in order to deliver the sort of performance required when dealing with near-realtime events. But neither StatsD nor Riemann currently supports a "high availability" topology. In other words, if a single Riemann or StatsD node fails, you will lose some data. The amounts of data lost may well be small—it could be as little as a the data captured over a single second. The author of Riemann is working on a clustered model, but at the time of this writing, this isn't yet available. So if you deploy either of these systems, you will at best be aiming for a warm-standby, where you can fail over traffic to a standby node in the event of failure.

Pick Your Protocol

Most of the systems we have talked about so far allow communication via a number of different networking protocols, the two most common being UDP and TCP. Why the choice? Simply put, TCP gives guarantees about the packet being sent actually being received, whereas UDP doesn't, though UDP has the upside of being faster. Note that the guarantees around TCP are not the same thing as guaranteed delivery and fulfillment—that is something that an application or a higher level protocol needs to provide (although they will often use TCP under the hood to achieve this). UDP, on the other hand, is a faster protocol that will occasionally lose packets. The performance improvements you can see and the amounts of data lost will vary from platform to plat-

form, but you will typically expect to see a significant reduction in latency when using UDP over sending the same traffic via TCP.

Lots of different networking protocols exist, and they all have their own tradeoffs. Often you will be limited in your choices by the underlying platform—for example, virtualized environments like Amazon Web Services will often prohibit multicast protocols. But your choices are limited mostly by what your tools support.

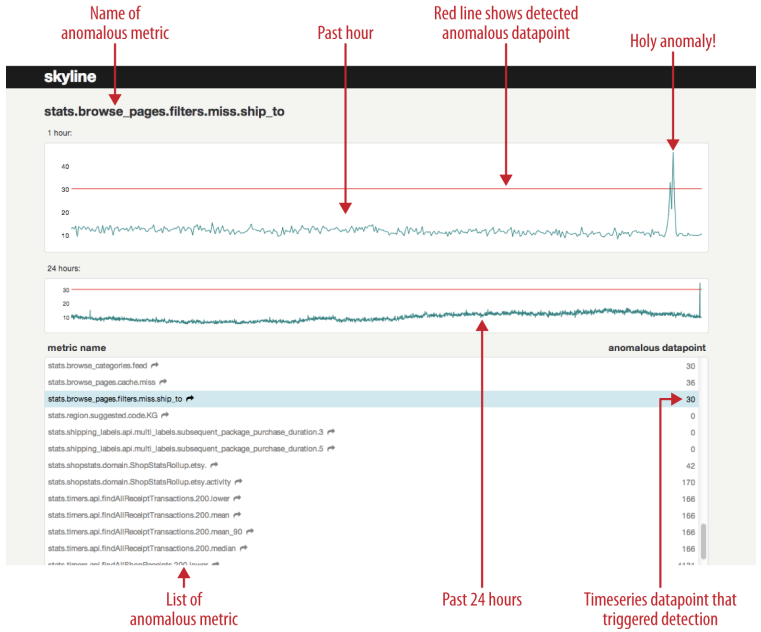
So when given the choice, which should you use? When getting a large amount of data from multiple machines, you can often afford to lose a few events here and there—what does it matter if I lose a CPU measure from a host if I'll get another one in 10 seconds? That would probably mean that UDP would be good enough. Likewise, if I miss the fact that an order has been placed, what is the impact? If I'm just recording the fact to get a general sense of how the site is performing and to carry out some general correlation analysis, then that's probably OK, especially if a separate, more accurate offline reporting system handles the detailed data. Otherwise, I may want to consider TCP or even using something like AMQP so I can take advantage of queues like RabbitMQ, which can provide guaranteed delivery. What guaranteed delivery actually means and how queues and asynchronous systems work is a big topic; a good starting point would be Gregor Hohpe and Bobby Woolf's book Enterprise Integration Patterns (Addison-Wesley, 2003).

You will often need to make tradeoffs between speed and accuracy. When dealing at scale, it's very hard to achieve both. Knowing what is more important for you will help you make better choices. Often, the answer is that you want both—speed so you can react now, and accuracy later. Consequently, different systems will be used to convey the same information, resulting in different numbers for the same thing. This means you may have to get very good at handling how data is shared with different parties and explaining why the differences might exist.

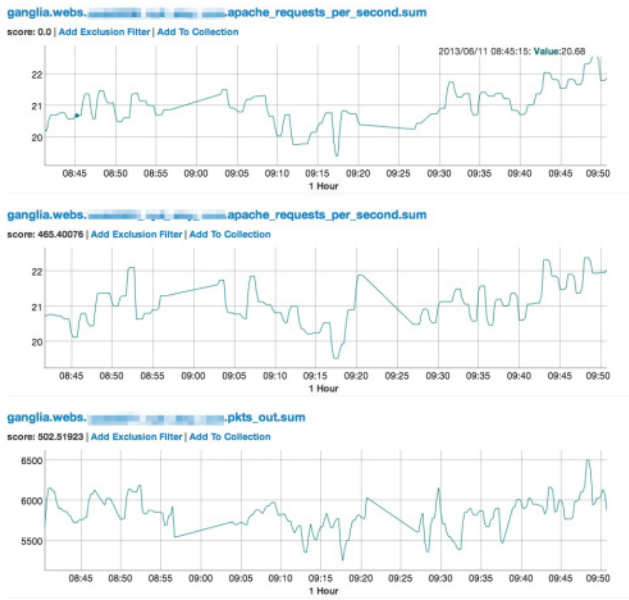
Anomaly Detection—Skyline and Oculus

When we gather large amounts of data over a long period of time, we can get good at understanding patterns. What does good look like? What does bad look like? Can software help us here, too? But this can be a lot to take in—and can we really comprehend hundreds, thousands, or even millions of metrics that are flying around? Etsy has

developed two open source tools that can help in this space: Skyline and Oculus.



In operation, Skyline sits in the background looking at metrics, trying to spot odd patterns. When it identifies them, it flags them, allowing a human being to take a look to determine if it's something of concern. Oculus is a system that then lets you search to see if the anomaly has happened in the past:



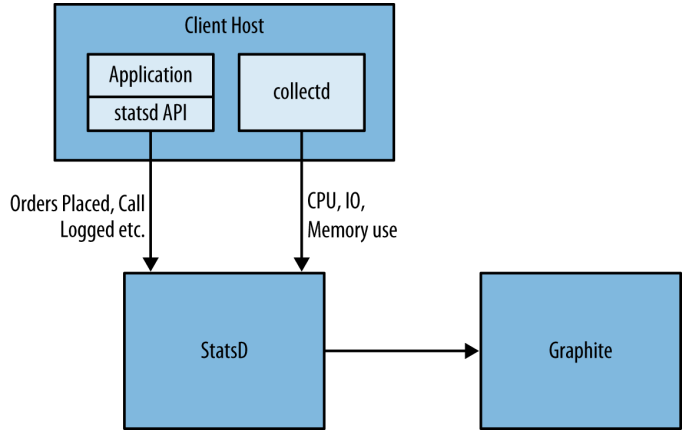
As of the time of writing, Skyline and Oculus are very new systems, and Etsy is quite open about the fact that the system is too noisy for it to be used as an active alerting system. Instead, they recommend it be used as a tool to provide an additional source of data that can help improve your own understanding. It is certainly early days for these systems, but it's a fascinating space that will no doubt continue to mature.

Getting Data In

Many of the tools we've covered make use of a plethora of supporting subsystems to get data into them. Collectd, for example, is the de facto standard in pulling operational metrics out of Linux systems, and it also supports its own plugin module so it can extract data from common Linux programs like Varnish or Apache. Support exists to get this data into Graphite and even StatsD. But as we said at the beginning, what we're looking for is a way to get access to non-IT operational data in a realtime fashion. So how do we do that?

Several of the tools we outlined have very simple, text-based schemas. This makes interoperability a fairly trivial affair. APIs now exist for many platforms that allow people creating applications to send their data to these reporting systems. A good example of this is the Metrics

library from Codahale (a non-trivial library to Google for!). Codahale's Metrics library allows JVM-based systems to store counters, gauges, and the like in a fashion similar to StatsD. It then allows you to expose these values via JSON or JMX or send the information to remote systems like Graphite, which it supports out of the box (check). This makes it simple for an individual service instance to send business-related metrics (for example, orders placed, money taken, customers served) and have them stored and aggregated in a central location.



When you consider that you will probably also be receiving information from the machine it's running on, some interesting possibilities occur. You can now see, for example, how business-related metrics and operational metrics relate. Perhaps you can start to see that your CPU is proportional to the number of customers being served. That might help you make decisions about how to grow (or shrink) the number and size of machines being used. Or perhaps you see that after a new release of software, your response time has increased despite there being no additional orders placed, meaning that the new version of code may have introduced a performance problem. These sorts of correlations become possible when you have access to all this data at once.

Small and Perfectly Formed

All the tools outlined (with the possible exception of Riemann) have one thing in common: they're focused on a single purpose. They aim to do one thing and do it well. And they optimize for ease of integration. In many ways, they end up espousing the Unix philosophy:

> "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface." —Douglas McIlroy

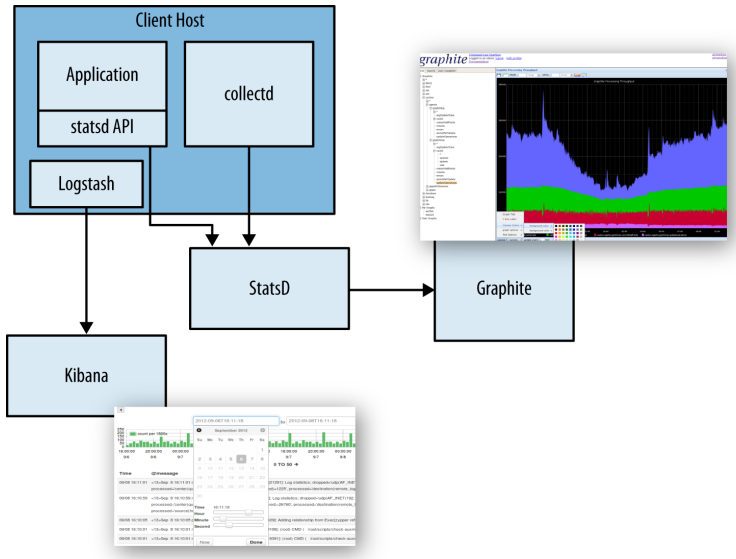
Due to their tightly focused nature but simple textual interfaces, tools like Graphite or StatsD allow themselves to be reconfigured in multiple different ways in much the same way as Unix programs like *ls* and *grep*. Not only that, but they also make it easy to swap components in and out as required.

Free, open source software typically comes about as a result of small groups of passionate people (or often individuals) working either in their spare time on projects or else extracting the work from larger systems built during their day jobs. By their very nature, the resulting projects tend to be small, highly focused pieces of software that have limited scopes. The best of them often embrace the Unix philosophy of "do one thing, and do it well."

On the other hand, tools that emerge from the commercial vendor space tend to take on a different form. The creation of these tools is typically the main focus of activity. Teams working in these environments have the luxury of time and support to build ever-larger systems, and often the way these tools are sold reinforces the desire to create a suite of products with lots of up-sell potential. These suites of products are built to work together, but more often than not, little thought is given to their potential to work with open source tools or tools from rival vendors.

It is for this reason that commercial products are often seen as large, unwieldy beasts. Their drive toward releasing suites of products from the same vendor tends to trump concerns about ease of interoperability, which often leads to a glass ceiling in our use of them. ("If only I could swap out this bit here and use this thing instead, I could solve all these problems!")

A Confusing Landscape



If we were to adopt many of the tools here, we would end up with a bewildering array of data flowing around our systems. Tools like StatsD and Riemann let us aggregate and filter these events in a fairly generic way. Graphite gives us places where we can at least centralize this information, but by then it has travelled over different protocols, relayed by different daemons. Ultimately, all this data can be summarized down to a simple event structure:

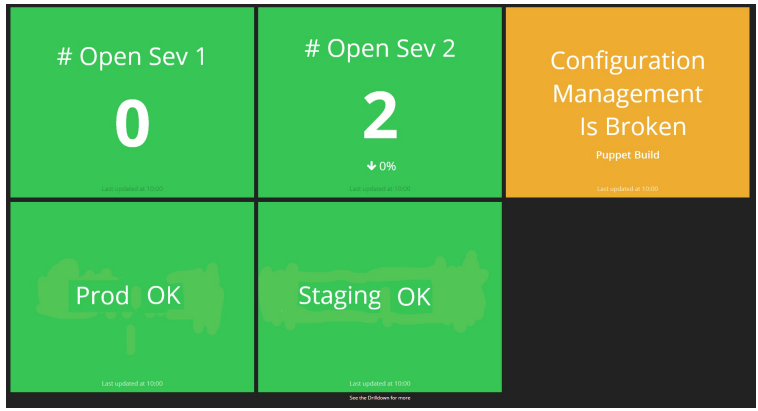
- Timestamp
- Value
- Description

If you look at the protocols for the tools we described earlier, you can see these fields in nearly all of them. In StatsD, the timestamp is automatically stored when the event is received. Graphite uses this format. So could we unify the protocols used for these events to simplify the flow? And do we need to? Whether or not a de facto standard emerges for event plumbing, one thing seems certain: we'll have more choices rather than fewer in the near future.

Reaching Your Audience

Whether or not the aggregation, filtering, and relay activities that get data where it needs to be ever end up in a more standard, generic interface, there is one area that should be actively discouraged from becoming one size fits all, and that is the display of data. Different audiences want different things in different contexts.

You have the screen in the corner of the room, where you probably want a small number of large, visible numbers and colorful displays to attract attention. Then there is the screen you'll go to when an alert is triggered to see what just happened. Then there is the fluid reporting environment you'll want to start looking for trending information to understand where your next problem might come from.



When building displays for this purpose, understand who needs to see what, understand in what context they will be receiving the information, and work out from them which actions they will want to take in order to do their jobs. There are a number of tools out there that can help; we've already covered Graphene and Graphite's own basic dashboard capability, but you also have tools like Dashing, which is fast becoming a stable of many big visible displays.

Conclusion

The continuing emergence of Open Source tools in the space of monitoring, data collection and aggregation, and analysis has lead to more and more organizations building their own bespoke systems from a collection of small, highly focused open source tools. Think of the use of StatsD sending data to Graphite. Both tools are focused in their use

cases, but the underlying protocols used to communicate with them are so simple that interoperability is easily achieved. Another hat tip goes to the Unix philosophy here ("write programs to handle text streams, because that is a universal interface"). Riemann seems to delight in the possibility of doing anything with the events it receives, prioritizing extensibility perhaps at the cost of simplicity, but unarguably putting lots of power at our disposal.

Where the development cycles of large suites of products are slow, new and interesting highly focused Open Source applications and systems are emerging all the time. Those organizations adopting this approach to gluing together subsystems with simple protocols find themselves able to more easily take advantage of new tools as they emerge. A new, better log-shipping tool emerges? Well, if we can create an adapter for our existing log search tool (like Kibana or GrayLog 2), then off we go!

No longer do we need to hide our data away or wait weeks to find out what it can tell us. Using freely available tools, we can free up this information and have it flow as events across our networks, and we have the tools at our fingertips that let us harness these events and derive the understanding that will make our systems better, our companies more successful, and our customers happier. Go get started!

About the Author

Sam Newman is a technologist at ThoughtWorks, where he currently splits his time between encouraging and sharing innovation globally and helping design and build their internal systems. He has worked with a variety of companies in multiple domains around the world, often with one foot in the developer world and another in the IT operations space. If you asked him what he does, he'd say, "I work with people to build better software systems." He has written articles, presented at conferences, and sporadically commits to open source projects. He is currently writing a book on building Microservices, which should be available from O'Reilly in the fall of 2014.

O'REILLY®

Wait. There's More.

4 Easy Ways to Stay Ahead of the Game

The world of web operations and performance is rapidly changing. Find what you need to keep current at **oreilly.com/velocity**:

More Reports Like This One

Get industry intelligence in timely, focused reports written to keep you apprised of the current and trending state of web operations and performance, best practices, and new technologies.

Videos and Webcasts

Hear directly from some of the best minds in the field through free live or pre-recorded events. Watch what you like, when you like, where you like.

Weekly Newsletter

News happens fast. Get it delivered straight to your inbox so you don't miss a thing.

Velocity Conference

It's the must-attend event for web operations and performance professionals, happening four times a year in California, New York, Europe, and China. Spend three supercharged days with the best minds, companies, and people interested in the same things you are. Learn more at **velocityconf.com**.