iVoyeur
Changing the Game

DAVE JOSEPHSEN

Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is senior systems engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.
dave-usenix@skeptech.org

At my last job, the Windows sysadmin would plan, for every new software implementation, sufficient time to install and configure the application a minimum of three times. This was not padding—they actually installed and configured every new application the company brought in three times. For more complex applications they would rebuild more often. I seem to recall that they rebuilt "Documentum" [1] something like nine times.

I thought it was kind of crazy at the time, but, looking back, I think I can, if not relate, at least understand. With technology, there is some operational coefficient of long-term success that defies prediction. Sometimes you don't know what you're not going to like about a solution until you've installed it. Sometimes proper integration into the existing environment isn't obvious until a solution is improperly integrated. Sometimes you need to throw a few things at the wall to see what sticks, and sometimes you need to break a thing, to see what can be made of the pieces.

I don't know if this property has a name, but I get that it's there. The Windows guys at my last job built and rebuilt to tune their systems for this variable in a kind of institutionalized brute force attack. They did this every time (probably without being able to articulate exactly why) partly because their choices were limited and partly because that's just the kind of thing they do. I think the reason I (and probably you) find their technique questionable is that, to some extent, optimizing for this property is the meat of what professional system administrators do; that this is what it means to hone our craft. We strive to excel at solving for elegance. A lot of the time, even when we get it right something new will come along that makes us rethink our architecture. Game changers create new possibilities, and our solutions might need to change to encompass them.

I can remember reading the first papers on RRDTool and considering the options for Nagios integration. Various Nagios plugins (as you're no doubt aware) pass back performance data along with their normal output. The way Nagios deals with this performance data is configurable. The shortest path is to configure Nagios to send performance data to a tool that parses out the metrics and loads them into RRDs. Several tools in various languages exist to do this, such as NagiosGraph [2] and PNP [3].

This light glue-layer between Nagios and RRDTool seemed elegant. You were using data you already had in a new way. The regex-based parsing gave you performance graphs across all of your hosts and services for just a few lines of configuration,

RRDTool lends itself to exactly this sort of data exportation, and you were using hooks built into Nagios to make it happen. For the price of a Perl script (or whatever) and a few lines of configuration, you'd just bought yourself performance data for every monitored service.

The problems become apparent at around 350 hosts for most modest hardware. It just doesn't scale well. Even for small installations, Nagios isn't going to get anywhere near real-time data; it's intended to operate on the order of minutes, not seconds. Accompanying this realization was a second: namely, that you couldn't easily scale the system horizontally by adding more hardware, nor could you bolt on a new solution in a way that would make it easy to display the data gleaned from both Nagios and the new stuff. Nagios and RRDTool had been tied together for better *and* for worse.

For most sysadmins, Cacti [4] and/or Ganglia [5] changed the game toward discrete systems for availability and performance monitoring. We had to go through and install new agents on all of our hosts, but we did it because these solutions (and Ganglia in particular) do a fantastic job of getting near-real-time data from a massive number of hosts with very little overhead. This also seems elegant, but there are still several problems. For one, Ganglia assumes a cluster model, which is a handy assumption that helps us combine and summarize data, but also forces a dashboard view of our environment that may not always be optimal. For another, it's still difficult to mix and match the data from different sources. If I want to graph something new, then I'm going to have to send it through Gmond or use a different front end to do my graphing.

It seems odd to me, given the problem RRDTool was intended to solve, that taking data from different places and storing it together in such a way that a generic front-end can graph it is this difficult. That last sentence could just about be a reworded mission statement for RRDTool, and yet nearly all the tools we've built on top of it are purpose-specific. I've long thought that the folks writing the front ends were just not thinking big enough, but now I'm beginning to believe this isn't accidental.

Some of the problem might have to do with the way RRDTool itself is architected. Different data sources and types of data, and even different intended uses for the same data imply different RRD requirements. For example, the heartbeat for a metric collected from Ganglia is going to differ wildly from one collected from Nagios. Any higher-level tool must make some assumptions and provide sane defaults, and while I don't think it's impossible to write something on top of RRDTool that could deal with a much larger set of assumptions, I have to admit that RRDTool's configuration rigidity is encouraging the higher level tools to be purpose-specific to some degree.

To have a more source-agnostic storage layer, it would be easier if we had something akin to RRDTool that was more relaxed about how often data was updated, and less concerned about categorizing it into pre-defined types.

Enter Graphite [6].

Graphite was developed internally by the engineers at Orbits.com and changes the game all over again. The name actually refers to a suite of three discrete but complementary Python programs, one of which is itself called "Graphite" (I assume they did this to make it more difficult to write articles about Graphite).

The first of these is Whisper, a reimplementation of the RRD format that makes the modifications to the data layer I mentioned above. Whisper does not particularly care how far apart your data points are spaced, or, indeed, if they arrive in sequential order. It also does not care what kind of data it is internally. Whisper stores all values the same way RRDTool would store a "Gauge" data point.

Data interpretation is handled by the front end using various built-in functions that modify the characteristics of the data when it's displayed. For example, at display time, the user runs the "derive" function to obtain a bytes-per-second graph from byte counter data stored in Whisper in its raw format.

The critically important upshot is that by making the storage layer agnostic to data type and frequency, new Whisper databases may be created on the fly with very little pre-configuration. In practice, the sysadmin specifies a default storage configuration (and, optionally, more specific configurations for metrics matching more specific patterns), and after that all Whisper needs to record a data point is a name, a value, and a date-stamp.

Carbon, the second Python program, listens to the network for name/value/date-stamp tuples and records them to Whisper RRDs. Carbon can create Whisper DBs for named metrics that it has never heard of, and begin storing those metrics immediately. Metric names are hierarchal from left to right, and use dots as field separators. For example, given the name "appliances.breakroom.coffee.pot1.temp", Carbon will create a Whisper DB called temp in the $WHISPER_STORAGE/appliances/breakroom/coffee/pot1 directory on the Graphite server. Carbon listens on TCP port 2003 for a string of the form "name value date". Dates are in EPOCH seconds. Continuing the coffee pot example, I could update that metric with the value 105 with the following command line:

```
echo "appliances.breakroom.coffee.pot1.temp 105 1316996698" | nc -c <IP> 2003
```

I passed -c to netcat so that it wouldn't hang waiting for a reply from Carbon. Obviously, you need the netcat with -c support to do that (http://netcat.sourceforge.net). Most large Graphite installations front-end Carbon with a UDP datapoint aggregator, but more on that later. The critically important upshot of this is that there is a socket on your network to which anyone (with access) can send data and have it stored and ready for graphing immediately. Whisper's data agnosticism and Carbon's network presence combine in such a way that data collection and presentation is no longer an ops-specific endeavor. For example, Carbon clients have been written for about every popular programming language out there, making it trivial for developers to build applications that send interesting metrics to the Graphite server. There's no reason why the security guys couldn't tie in their snort stats and/or logsurfer instances for that matter.

In his now famous blog post "Tracking Every Release" [7], Mike Brittain shares how the engineers at Etsy have their deployment tool sending a 1 to their Graphite server every time a code deployment takes place. Since Whisper doesn't care about data frequency, it's possible to graph instances of things like deployments that only happen every so often. At Etsy they superimpose these data points as vertical lines over other metrics to correlate events, such as PHP warnings per second, to code releases.

Finally, Graphite is the Web front end to ... well, Graphite. Graphite runs on the Apache Web server with mod_python, and includes a novel Web-based command-line interface (with tab completion) that makes it easy to create on-the-fly graphs

from any combination of stored metrics. It also has a tree view reminiscent of Cacti, and a user-configurable dashboard view. My favorite piece of the front end is the URL interface, which allows the creation of graphs by specifying URLs. This feature is something I've been wanting for a long time. It enables integration with just about every monitoring system out there, including Nagios via its "action_url" attribute.

This seems elegant. We've certainly come a step closer to separating the polling engines from the storage engines from the display engines. It's unfortunate that, once again, I'm looking at a single application that is storing and displaying the data, but I think this has more to do with the lack of independent front ends that support the Whisper data store than any intrinsic dependency between the two. Graphite changes things. It introduces new possibilities. I plan to write a few more articles exploring Graphite, its installation and usage intricacies, and especially the integration possibilities. So stay tuned.

Take it easy.

References

[1] EMC Documentum: http://www.emc.com/domains/documentum/index.htm.

[2] NagiosGraph: http://nagiosgraph.sourceforge.net/.

[3] PNP: http://docs.pnp4nagios.org/pnp-0.4/start.

[4] Cacti: http://www.cacti.net/.

[5] Ganglia: http://ganglia.sourceforge.net/.

[6] Graphite: http://graphite.wikidot.com/.

[7] Mike Brittain's "tracking every release" post: http://codeascraft.etsy.com/2010/12/08/track-every-release/.