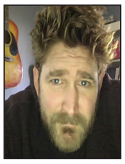
iVoyeur
Spreading

BY DAVE JOSEPHSEN



Dave Josephsen is the sometime book-authoring developer evangelist at Librato .com. His continuing mission: to help engineers worldwide close the feedback loop. dave-usenix@skeptech.org

In engineering, we are told to avoid repeating ourselves [1], but as a blogo-vangelizer (or whatever it is I'm doing now), I find it an increasingly burdensome and self-defeating mantra. It'd be great if I could give one talk and consider the subject of that talk closed. However, over the course of my first year as a developer evangelist, wherein I've delivered 12 conference talks, I've slowly begun to realize two very interesting facts.

First, most of the people who came to the conference don't see my talk. Even if the conference is a single-track, many attendees are consumed by a fire at work or by some really interesting "problem solving" (read: cat gifs), or they're in the hallway talking to the speaker from the last session. Whatever the reason, only a fraction of the attendees actually attempt to parse my one-two punch of words and slides.

Second, I very often fail to convey what I intend to the fraction of attendees who actually listen to me. I know this because when I talk to people who attend my talks, our conversations often go something like this:

Attendee: "Hey, I really enjoyed your talk."

Me: "Awesome, thanks! I hope it helped."

Attendee: "It did! I'm going straight home to <do horrifyingly wrong thing>."

Me: "Good god, why?!"

Attendee: "Well, silly, because you said <understandable but horrifyingly wrong interpretation of thing I said that would take me days to unravel and correct>."

Me: "Yeah, I can't take the credit for that. I actually copied it from <person who works at Microsoft>."

My point is, repeating yourself in an education context is not a bad thing (especially if you can't seem to get it right the first time). Many tech speakers riff on variations of the same talk over and over again for years. I used to suspect this was laziness, or that they'd gotten trapped by their own cult-of-personality, but now I'm realizing that you have to repeat yourself a lot to actually reach a critical-mass of mind-share in this medium. This is good news for me, because it's pretty often the case when I find myself belaboring a point—writing and talking a lot about the same subject—that it's because I'm trying to share something I wish I would have understood years ago.

Lately, I've been writing a lot about fat data points, which is the data storage format employed by Librato in our metrics product, and it's certainly the case that I wish I'd have understood them years ago. At Librato, a common use case for us is that of service-side aggregation. This is the practice of customers emitting measurements to us directly from inside worker threads running across lots and lots of geographically dispersed computers.

If a customer spawns ten thousand worker threads, and each of them emits a few measurements, we can easily wind up with upwards of fifty thousand in-bound data points in the

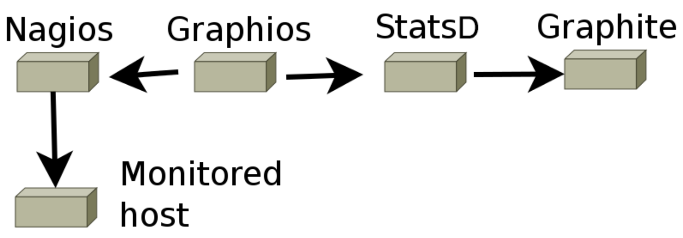


Figure 1: My tool-chain for the purposes of this article

space of a second, which we then need to aggregate in a statistically significant way. Taking the average of a set this size almost certainly destroys the truth hidden within the data, so for this, and many other reasons [2], we use fat data points to preserve the truth.

When writing about fat data points became talking about them at LISA14 [3], I got a pretty awesome question from Doug Hughes. It was simple, direct, and conveyed a deeply satisfying sense that I'd managed to successfully communicate the concept. Specifically, Doug's question was: "Okay, but how can WE use this?"

Avoiding the obvious (and correct) answer, that you should replace whatever you're currently using with Librato as soon as possible, it's actually possible to preserve spread data today with systems like RRDtool and Graphite. So in the interest of giving a meaningful answer to Doug's question, I'd like to show you how you'd configure Graphite to preserve spread data—the sum, count, min, max, average, and etc.

For the purposes of this how-to, I'm using a Nagios system that's emitting metrics to Graphite by way of StatsD. The metrics-extraction from Nagios is being performed by Graphios [4]. I'm going to use the one-minute CPU load metric as my example since I'm lazy and unimaginative. Figure 1 is a quick-and-dirty sketch of my setup.

Graphite controls rollups with the storage-aggregations.conf file. When a new metric is discovered for which there is no existing Whisper database, Carbon attempts to match the metric name against the rules in storage-aggregations.conf, beginning at the top and continuing to the bottom. The first line that matches the metric name wins, and no further lines are parsed once a match is found. If you're really paying attention, then you've probably realized that these rules make it impossible to assign different consolidation functions to different archives inside a Whisper file.

In order to maintain, for example, both the min and max values for a series in Graphite, therefore, we need to feed Graphite the same metric with two different names. That way we can match each variation of the metric name to a different rule in storage-aggregations.conf.

One simple way to do this is via StatsD's *timer* data type [5]. StatsD timers are intended to time things like function calls, to see how long they take to execute, but in practice you can use a timer to measure anything you might otherwise use a *gauge* to measure. The primary difference is that where passing a gauge into StatsD will merely result in a single value, a timer will cause StatsD to compute and emit a whole slew of interesting summary metrics, including the min, max, sum, count, and even percentiles for the StatsD flush interval.

So my strategy here is to emit the one-minute CPU load as measured by Nagios into StatsD as a timer. Then I'll configure storage-aggregation rules in Graphite to match the min, max, sum, and count for the summary statistics emitted by StatsD. When I'm done, I'll have different Whisper databases for this metric for each of the summary types I want.

Beginning in the Nagios configs, I'll configure a custom object variable called *metrictype* in the service definition of the metric I want to preserve spread data for:

```
define service{
   use                  generic-service
   host_name            awacs
   service_description  LOAD
   check_command        check_load!50,60,70!80,90,100
   _graphiteprefix      Piegan-Nagios
   _metrictype          timer
}
```

Graphios will parse out the `_graphiteprefix` and `_metrictype` custom variables, appending my prefix to the metric name, and translating the "timer" keyword into the associated StatsD wire-protocol [6]. On my system (hostname: `awacs`), this is what Graphios puts on the wire for StatsD:

```
Piegan-Nagios.awacs.load1:0.080|ms
```

No special configuration is required for StatsD. By default, StatsD will prepend two additional prefixes to my metric name: `stats` and `timers`. Here's what StatsD puts on the wire for Carbon:

```
stats.timers.Piegan-Nagios.awacs.load1.sum 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.sum_90 0.080
1416803719
stats.timers.Piegan-Nagios.awacs.load1.lower 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.upper 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.upper_90 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.sum 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.sum_90 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.count 1 1416803719
stats.timers.Piegan-Nagios.awacs.load1.count_ps 1 1416803719
stats.timers.Piegan-Nagios.awacs.load1.mean 0.080 1416803719
stats.timers.Piegan-Nagios.awacs.load1.median 0.080 1416803719
```

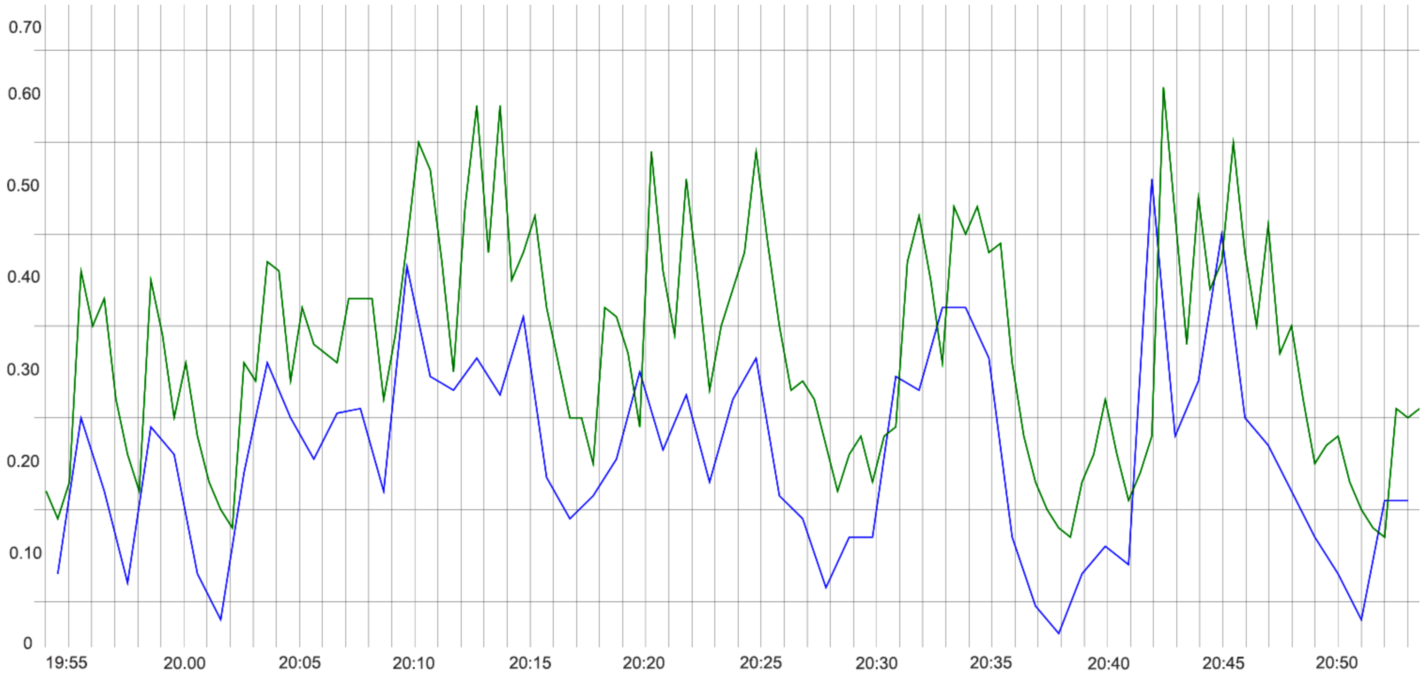


Figure 2: Plotting average vs. max for the same metric

To be clear, what's happening here is StatsD is accepting the load1 metric, and, because we've specified that it is a timer (the "|ms" suffix emitted by Graphios), StatsD automatically computes all of these summarization metrics across its flush interval. Most of these are self-explanatory; the metrics that look like thing_90 are the 90th percentile for thing (i.e., it is a number that 90 percent of the measurements in the flush interval are less than). `Count_ps` is the count divided by the number of seconds in StatsD's flush interval (literally, `ps` here stands for per second).

Moving to the Graphite side, I've added rules to match each of these StatsD summary metrics to my /opt/graphite/conf/storage-aggregations.conf file:

```
[min]
pattern = stats.timers.*lower$
xFilesFactor = 0.9
aggregationMethod = min

[max]
pattern = stats.timers.*(upper|upper_90)$
xFilesFactor = 0.9
aggregationMethod = max

[sum]
pattern = stats.timers.*sum$
xFilesFactor = 0.9
aggregationMethod = sum

<snip>
```

Carbon will use this file to properly create the Whisper databases for these metrics in a way that properly aggregates the data over time, preserving what's important to us. I can verify it's working by checking the creation log:

```
23/11/2014 04:43:26 :: new metric
    stats.timers.Piegan-Nagios.awacs.
    load15.upper_90 matched aggregation schema max
```

Or by running `whisper_info` directly against the DBs:

```
root@precise64# for i in *; do
> echo ${i}: $(whisper-info ${i} | grep aggre) ; done
count_ps.wsp: aggregationMethod: count
count.wsp: aggregationMethod: count
lower.wsp: aggregationMethod: min
mean_90.wsp: aggregationMethod: average
mean.wsp: aggregationMethod: average
median.wsp: aggregationMethod: average
std.wsp: aggregationMethod: max
sum_90.wsp: aggregationMethod: sum
sum.wsp: aggregationMethod: sum
upper_90.wsp: aggregationMethod: max
upper.wsp: aggregationMethod: max
```

At this point, perhaps obviously, I can craft a graph depicting the difference between the average and max rollups (Figure 2).

An interesting side effect of using StatsD timers this way is that you can also set up custom storage schemas for different types of spread data. For example, you could keep 10-second resolution on the mean and median values for 24 hours, and toss them after that while preserving the count and sum metrics at 10-minute and one-hour resolutions for years (since those rollups are effectively lossless and enable you to accurately compute the average at display time using the `divide()` function).

With a little thought, you'll wind up with a metrics storage system that far more accurately reflects your data, while making very effective use of space on disk. As always, I hope you found this useful in your quantification endeavors, and I highly recommend the use of spread data to protect the long-term fidelity of your beloved measurements.

Take it easy.

References

[1] "Don't Repeat Yourself": http://en.wikipedia.org/wiki/Don%27t_repeat_yourself.

[2] "Sensical Summarization for Time-Series": http://blog.librato.com/posts/time-series-data.

[3] LISA14: https://www.usenix.org/conference/lisa14.

[4] Graphios: https://github.com/shawn-sterling/graphios.

[5] StatsD Metric Types: https://github.com/etsy/statsd/blob/master/docs/metric_types.md.

[6] StatsD Line Protocol: https://github.com/etsy/statsd/.