iVoyeur

Changing the Game, Part 3

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

During the riots in London (the most recent ones), a friend pointed out that the top-selling items at Amazon.co.uk (as measured by their own "movers and shakers" page) were baseball bats. Before that, if you were to give me a data dump of all of Amazon's sales data for the past whenever, I confess I'd probably be at a loss for what to do with it. I would know that the data set held fascinating economic and sociological truths, but I wouldn't know the questions to ask to tease them out off the top of my head.

Given, however, the baseball bat tidbit and the accompanying revelation that the second highest selling item was baseballs, my head virtually spins with conjecture. I imagine my own city aflame outside my hastily boarded up windows. The shouts of looters and the screams of car-alarms penetrate my makeshift barricade while I click the Next-Day Air option on my order of a Louisville Slugger. My heart goes out to those people, truly, but what *were* they thinking? What UPS-man in his right mind drives a delivery truck through hordes of looters? Were they really so proper that they felt the need to buy some balls to keep up appearances while their city burnt down around them? Were the recipients of these orders even in London? Perhaps it was the surrounding burgs making preparations just in case?

At any rate, that little bit of knowledge makes me look at the larger data set with new eyes. It invites me to explore possibilities that hadn't occurred to me before, and, for me at least, that's the way it goes with data analysis. My imagination needs a bit of a kick in the shins to get it going. In the example above, we took our inspiration from the data itself, but it's also possible that a better understanding of, and easier access to, a few analysis techniques could inspire us to look at our data in new ways. Herein lies what I feel is the most fundamental difference between Graphite and RRDtool. The latter provides the means to perform whatever math we wish on our data before we graph it, while the former gives us a bunch of statically defined analysis functions that we may apply as we graph it.

Normally, I would argue for RRDtool's flexibility, but, in practice, Graphite's predefined functions do a much better job of kicking me in the shins and arousing my curiosity. In this, the last article in my series on Graphite, I'd like to share a few of these analysis functions with you, and hopefully show you how Graphite has me thinking differently about my data.

We'll start with the exception to the rule, the one area where Graphite is arguably more flexible than RRDtool: derivatives and integrals. As you probably know, RRDtool wants you to categorize your data into one of several types, including

"GAUGE" and "COUNTER." This is so that RRDtool can make some underlying assumptions about the data you're storing. For example, the reason RRDtool gives you a "COUNTER" data type is so it can automatically compute the derivative from counter data. So if you have a metric such as the number of packets passed through an interface, RRDtool will automatically take the derivative of this counter and provide you packets per second.

In Graphite, you'll recall that all data is stored the same way RRDtool would store a "GAUGE" data type, which is to say, Graphite just stores the raw data. This means that if you want to compute packets per second from a counter metric, you need to apply the "derive()" function to the data when you graph it. If this seems counter-intuitive, well, it is at first. At least I thought so before I became familiar with the rest of the functions, but first things first.

Graphite functions may be applied in the Graphite GUI via the "Graph Data" button in the graph composer, but now that I've been using Graphite for a while, I find it more expedient to work with the URLs directly. This is both because I type better than I click, and because the function names in the documentation don't exactly match those in the GUI, so one avoids the need to hunt around in menus by simply typing them into the URLs. To do this, I first get some data into the graph in the graph composer, then right-click the graph itself and select "copy link location," and then I simply paste the URL into a new browser tab.

Functions apply in a C-like manner, as you would expect, and most of them accept multiple metrics and even wildcards in lieu of lists. For example, I can apply the derive function to "some.counter.data" like so:

```
&target=derive(some.counter.data)
```

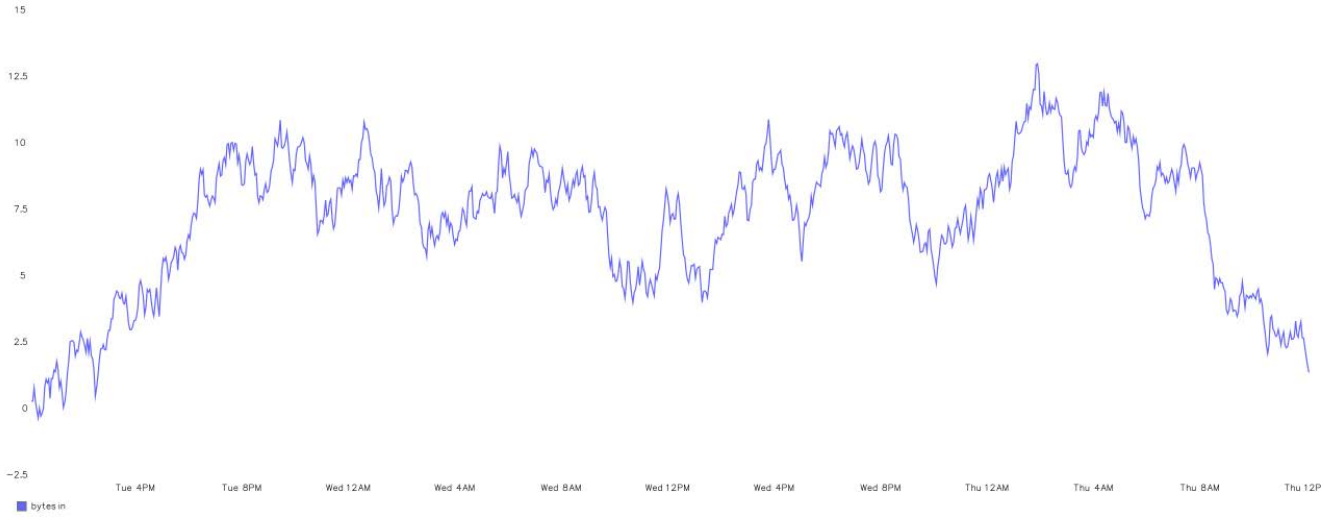

Figure 1: The derivative of a router's byte counter

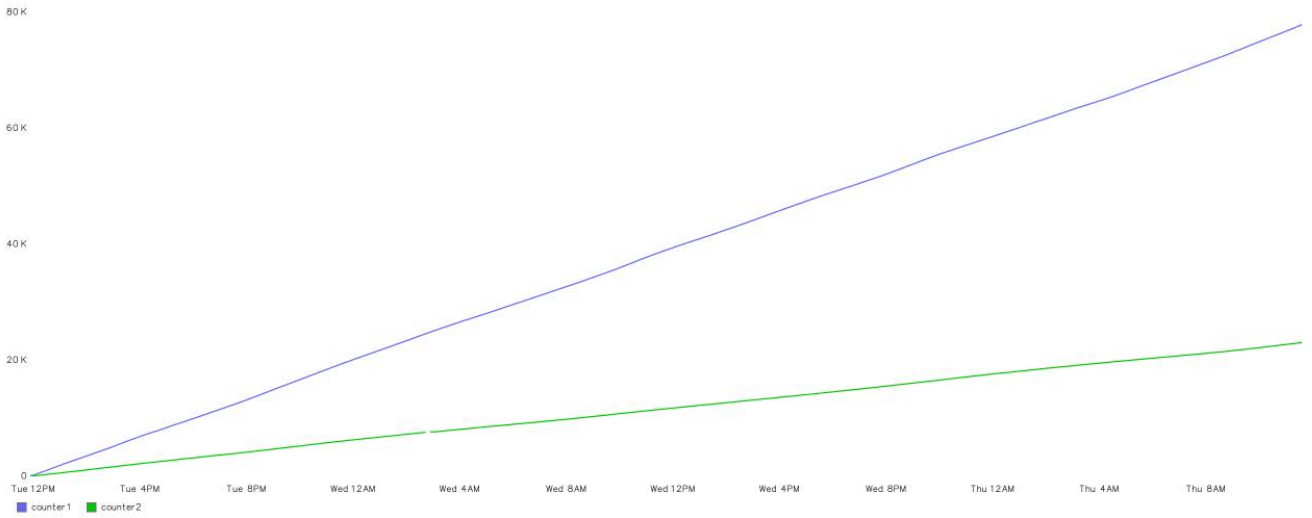

Figure 2: Raw byte count values from two routers

This function yields rate data as depicted in Figure 1, but that brings me to my first kick in the shins: namely, that sometimes looking at raw counter data is interesting. This wouldn't have occurred to me using RRDtool, but what if we compare the raw byte counters of two different routers, as seen in Figure 2? This could be useful capacity planning info, but it's not a fair comparison, because the routers have different total values, so one router will always appear to be growing at a smaller rate than the other. That's okay, Graphite provides us a "secondYAxis()" function, which easily allows us to draw one of these two data sets on its own Y-axis. So by graphing:

```
&target=router1.bytes&target=secondYAxis(router2.bytes)
```

we can get a clear picture of comparative rate of growth of the byte counters for these two routers, as seen in Figure 3. There's also an "integral()" function, which allows you to take GAUGE-based data sources and get counter-style data. If, for example you had a graph of widget sales per minute, you could apply the integral function to graph total sales for a given time interval.

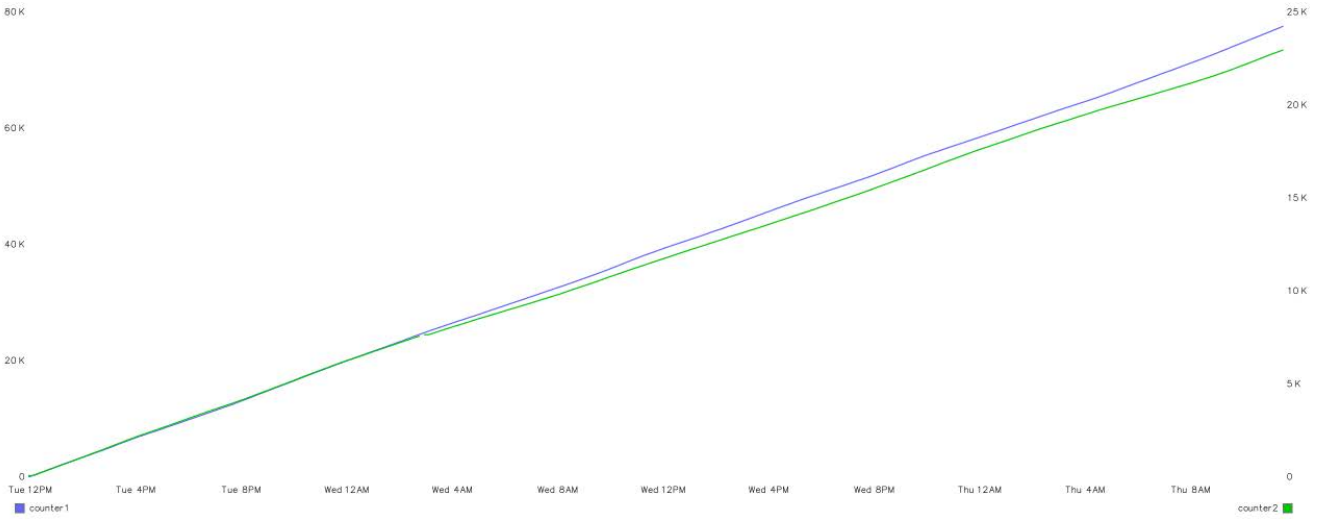

Figure 3: Raw byte counts from two routers compared with independent Y-axis

Now, if you're adept at RRDtool, take a moment and think about what it would have taken to "RRDtool graph" Figure 3, especially if you had been storing your counter data as type "COUNTER", as you should. It's probably possible, but I admit I don't know how to do it off the top of my head, and the idea of puzzling it out in Reverse Polish Notation somehow stops short of sounding appealing. Even if I did tease it out, I wouldn't be likely to apply the technique to other data sets for the benefit of my own curiosity, and the various RRDtool-based frontends out there wouldn't be much help to me in that endeavor. Graphite's functions invite me to visualize the data in new ways by virtue of their existence and accessibility. That's probably the biggest way Graphite has been a game-changer for me.

The functions themselves are fully documented at [1], and I can't cover all of them here, but let's take a look at some of my favorites, starting with "summarize()." This function allows you to re-compute the interval for a given set of time series data. So given a metric such as the number of users registering for an online service as depicted in Figure 4, we can imagine that the marketing team has a goal to maintain X registrations per hour and would like to display this data on a kiosk in the hallway, but they'll want it graphed as "registrations per hour" to reflect their goal. We can compute this graph, depicted in Figure 5, for them with:

```
&target=summarize(user.registrations,"1h")
```

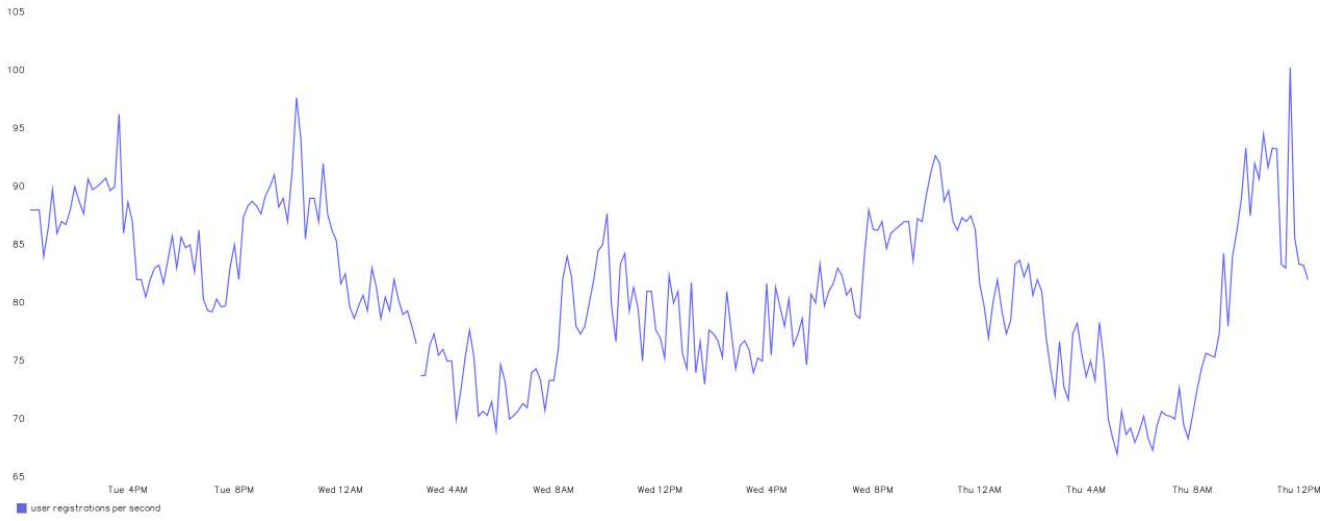

Figure 4: User registrations over time

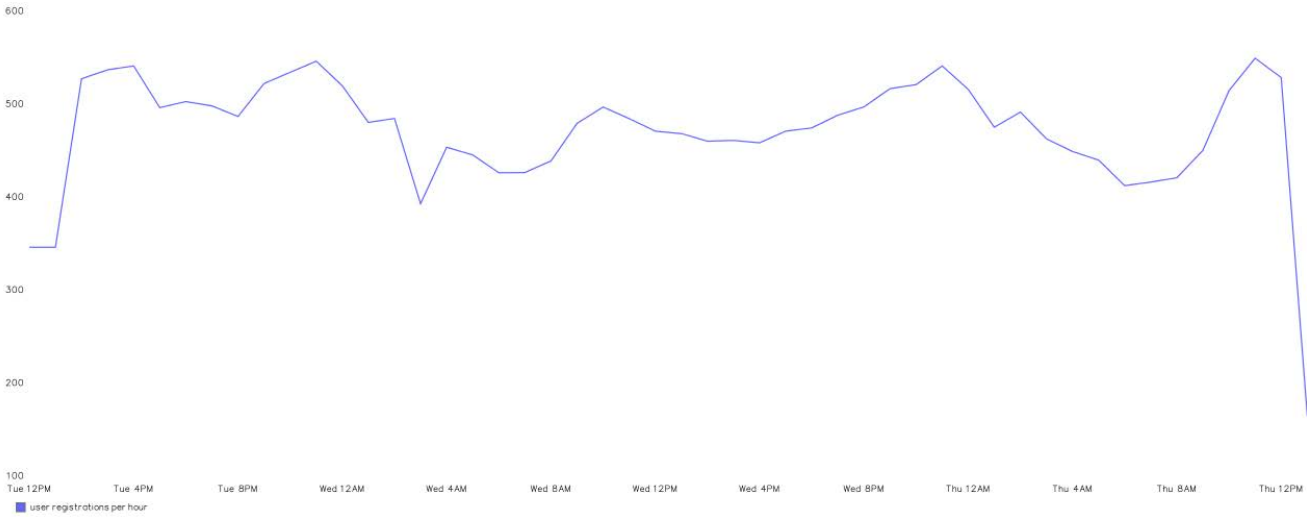

Figure 5: User registrations summarized hourly

To make their progress more obvious, we could add a horizontal line (constant) equal to their goal (Figure 6) with the "threshold()" function like so:

```
&target=summarize(user.registrations,"1h")&target=threshold(400,"Goal")
```

Figure 6: User registrations summarized hourly with a constant goal value

Functions are nestable, as in C, so we could add the data from last month to the graph by nesting the summarized target inside a "timeShift()" function. This would give the marketers some historical registration data from last month for context, while still maintaining a two-week period on the X-axis. This graph, drawn with the targets listed below, is depicted in Figure 7.

```
&target=summarize(user.registrations,"1h")&target=timeShift(summarize(user.
registrations,"1h"),"30d")&target=threshold(400,"Goal")
```

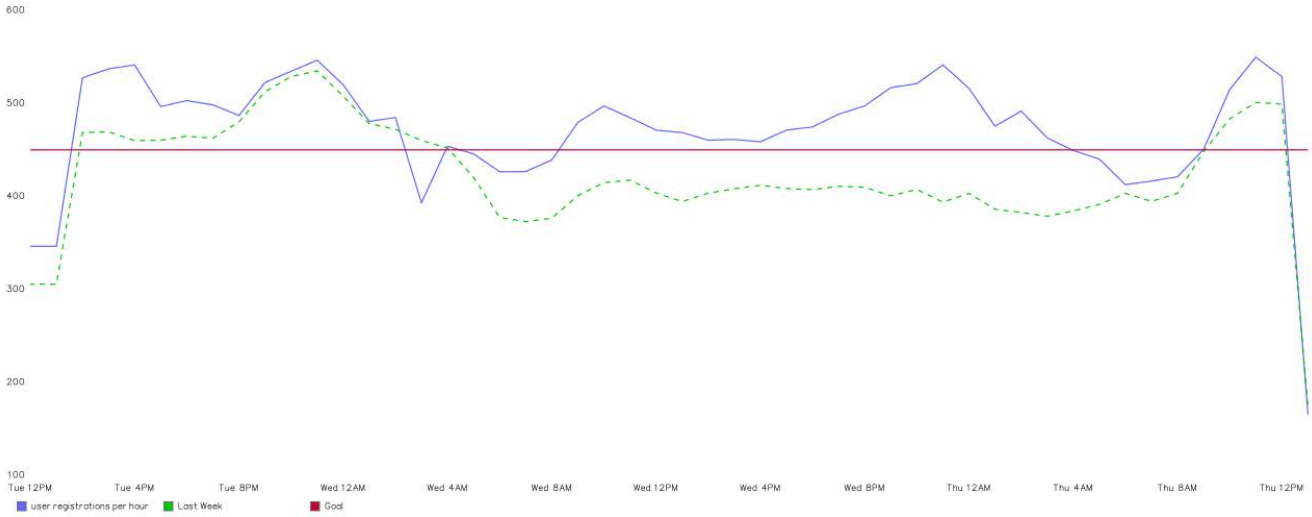

Figure 7: User registrations summarized hourly with a constant goal value and historical data

I really like the timeshift function. It's such an easy way to gain some context for almost any metric, and since I discovered it, every metric I graph seems to beg the question, "What was it doing last week at this time?" It's because of functions like this that Graphite feels more like an introspective tool and, by comparison, RRD-tool seems inflexible or perhaps even created for a different problem domain.

Various functions exist for combining multiple metrics into a single line: these are "sumSeries()," which creates a single line from multiple metrics by adding them together, "averageSeries()," which averages multiple metrics into a single metric, and "minSeries()" and "maxSeries()," which plot only the minimum or maximum value data points in the series. All of these functions support wildcards in the data-source field. For example:

```
&target=averageSeries(dc4.web.*.cpu)
```

plots a single line with the average CPU utilization of every Web server in dc4. Combinatorial functions are great for summarizing clusters or even datacenters. I find myself combining multiple averageSeries() of different metric types (CPU and disk, for example) using "secondAxis()." In this way I can get multiple metrics across entire datacenters on the same graph in a really usable way. Other functions exist for filtering individual metrics out of large lists. For example:

```
&target=highestCurrent(dc4.Web.*.cpu,5)
```

plots the CPU utilization of only the five currently most utilized Web servers in DC4. Combining these:

```
&target=averageSeries(highestCurrent(dc4.Web.*.cpu,5))
```

plots the average CPU utilization of the five currently most utilized Web servers in DC4. These are awesome for dashboards where you're just wanting to show things that are misbehaving, or aberrant behavior in general. I'm sure you get the idea by now. Although too numerous to offer a complete list here, filters include highest and lowest max, average, and current; filters which plot metrics that fall above or below static thresholds as measured by max, min, and average; and metrics that are most deviant from the rest of the series.

There are also a few advanced functions that bear mentioning. Included are functions for plotting the Holt-Winters Forecast, Confidence Bands (error bars), and Deviation. Holt-Winters is a statistical forecasting technique based on exponential smoothing. I wrote an article [2] about its inclusion in RRDtool, and I stand by what I said in that article: it's the coolest code that nobody ever uses.

I can't go into great detail here, but suffice it to say that the technique does a good job of predicting future data points based on existing data, even taking into account long-term and seasonal patterns (such as spikes or slow periods caused by human behavior on weekends and holidays). For many metrics, and especially system-based ones, problems can be detected by measuring their deviation from the "expected" value given to us by Holt-Winters, and Graphite makes this more accessible than it's ever been before.

Using Holt-Winters, I could create a dashboard that told us not only the five most utilized Web servers, but also their deviation, with:

```
&target=highestCurrent(dc4.Web.*.cpu,5)&target=holtWintersAberration(highest
Current(dc4.Web.*.cpu,5))
```

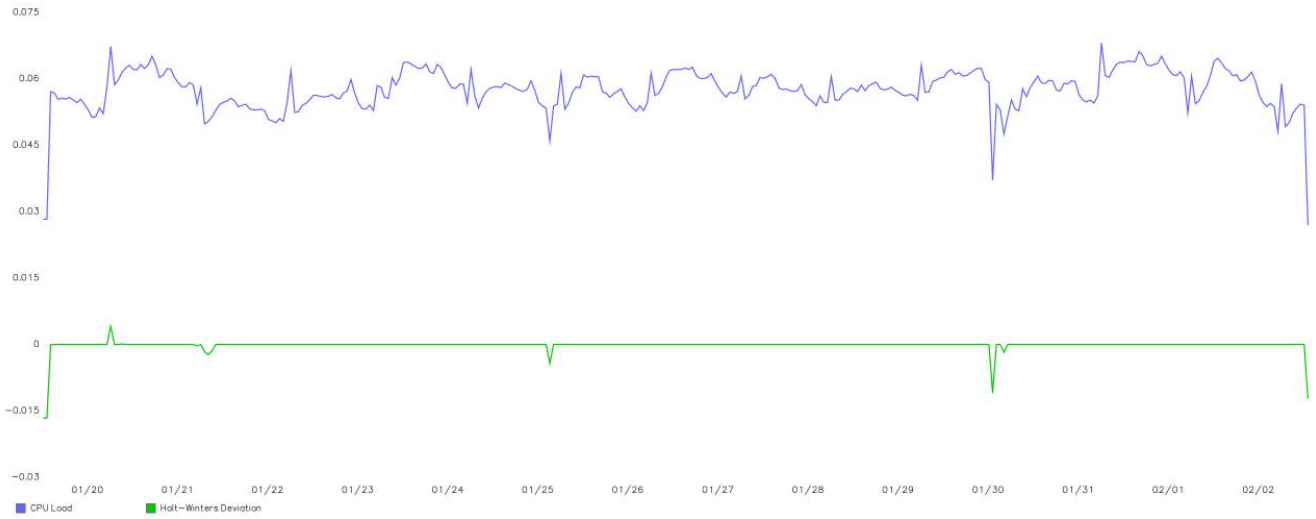

Figure 8: CPU utilization paired with Holt-Winters aberration

This graph might look something like Figure 8, where, while the lines on top would tell us what the CPU values were, the bottommost line would give us an indication of how "problematic" or at least how "unexpected" those values were.

There's a lot more to say here, but I'm afraid I'm at my word limit (to say nothing of having probably exhausted my Figures budget for all of 2012 (sorry, Jane-Ellen)). If my other articles on Graphite haven't convinced you to check out this truly excellent tool, I hope this last one has. I'm sure I'll revisit Graphite as it continues to mature, but I have so many excellent tools in the pipeline that I really must move on. Next time, expect the first in a new series of articles on a Nagios plugin that I'm really excited about called check_mk.

Take it easy.

References

[1] Graphite documentation—Functions: http://readthedocs.org/docs/graphite/en/latest/functions.html.

[2] Dave Josephsen, "iVoyeur: Hold the Pixels," *;login:*, vol. 33, no. 4, USENIX, 2008: https://www.usenix.org/publications/login/august-2008-volume-33-number-4/ivoyeur.