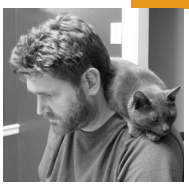
DAVE JOSEPHSEN

iVoyeur: 7 habits of highly effective Nagios implementations



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

ONCE, SOME YEARS AGO, WHEN WE were all younger, I had something of a desire for knowledge (I'm guessing you can probably relate). Don't get me wrong: today I spend a great deal of my time learning, more time even than when I felt that urgent want of it, but that's not how I'd describe it anymore. Now it's more of a habit, a taste for learning that is tempered by the things I wish I hadn't learned, tempered that is, by the knowledge that haunts me. I'll give you an example.

I don't remember who, and I don't remember where or when, but somewhere, somewhen, someone taught me the novel writer's "first line" rule. That the single most important part of a novel was its first line. That it was obvious from line one whether a piece of fiction was in fact a piece of crap. Now, I don't write fiction (or haven't yet anyway), and really I don't even consider myself a "writer," but this "first line" business, well, it haunts me. It's something I simultaneously wish I hadn't learned and can't let myself ignore. If you weren't aware of it, I'm sorry to have passed it on to you. Even when writing 2000-word articles for a tech journal, I invariably obsess over that confounded first line, revising this word or that, deleting entirely and starting over from scratch until my opening paragraphs have nothing whatsoever to do with my eventual subject matter. I'll give you an example.

The opening line the computer geek part of me wanted to begin this article with was, "There is a fine line between monitoring systems that are effective and those that are annoying and useless." "A fine line!?" the lit-geek in me exclaims. "Why don't you just define a CLICHÉ variable and put a 'while(1)' around it? Or better yet, send two sleeping pills to the entire subscribed readership of *;login:* since that's the effect you're obviously going for?!" (My inner lit-geek is kind of an elitist jerk.) At this point in my process, I usually delete the line in favor of something absurd and unrelated, but in this case I couldn't bring myself to delete it.

A fine line is in fact what it is; so fine that any one of the seven tips I'm about to share can completely change the effectiveness of an otherwise maligned Nagios implementation. Further, all of them are common pitfalls that I myself have fallen victim to at one point or another, so they are things that are likely to help other sysadmins. First line be

damned: this time I will have the courage to use a cliché if it is apt. This time I will NOT give myself over to the right half of my brain. This time, I REFUSE to write an unrelated yet entertaining introduction that requires a complex and clever segue into otherwise dry subject matter.

Oops.

1. Eliminate DNS Dependencies

The first tip I'd like to share has to do with name resolution. You can specify hosts in Nagios by IP or DNS name. It's probably a toss-up which of these is more reliable. Using names makes Nagios dependent on an operational DNS infrastructure. Using IPs eliminates the dependency but is a management nightmare; IPs will change, and Nagios won't be updated. In practical terms, using names gives you rarely occurring, large outages, while using IPs gives you more commonly occurring, individual outages.

You might think that Nagios being functional during a DNS outage would be the least of your worries, but you'd be wrong, in my humble opinion. If Nagios can remain functional during a DNS outage, it can provide good data on what boxes in the infrastructure actually ceased to function properly during the outage. There's a huge difference between a monitoring system that can provide good data during and especially after a cascading DNS failure and one that cannot. That's the kind of data that really lends credibility to the system, and much of the difference between effective implementations and maligned ones can be measured in credibility.

I recommend specifying names in the configuration files and then implementing a DNS cache and name resolver on the Nagios box itself. Set up zone transfers or otherwise automate the replication of DNS information from your real nameservers to the Nagios box. This solves all of our name-resolution woes; Nagios will point at itself for name resolution and won't rely on external DNS, while at the same time, there is no management overhead on keeping IPs up to date beyond the initial configuration.

We use djbdns [1] for this purpose, which I like very much. It's a small, lightweight, secure system that is easily implemented and updated.

2. Minimize Notifications

There are a couple of very large problems with monitoring systems that send too many notifications. The first is that the credibility of the monitoring system suffers when folks get notified about things they don't care about. It makes the system "seem" stupid, and that perception is going to make it difficult for you to get resources to improve the system.

The second, closely related problem is that people will begin to ignore the notifications. When actually important notifications are sent, they'll be ignored, and when/if management follows up, the monitoring system will be blamed for not sending notifications at all. The lack of credibility will make it easier for accusations like this to stick (despite evidence to the contrary). Further, if people don't trust the system, they'll be more likely to roll their own monitoring tools and less likely to ask you for help. This in turn will tend to magnify and compound the original perception until vendors are brought in and something truly stupid is implemented.

I'm not saying that you shouldn't *monitor* lots of services. I'm only saying that you should refrain from notifying anyone other than yourself about anything unless:

- they've specifically asked you to do it
- there's an SLA around it
- there's a policy requiring it

Even if one or more of these requirements has been satisfied, I'd offer them some alternatives instead (wouldn't a daily/hourly report of boxes with high CPU utilization be better?). Further, all notifications should be based on thresholds that you've performed some analysis to obtain. Holt-Winters forecasting [2] is superb for this sort of thing, but anything is better than nothing. The worst thing you can do is make up some arbitrary thresholds (or, worse, take theirs), create a notification group that includes everybody, and turn on Nagios (even if they ask you to).

There's a creeping entropy about this problem that makes it seem more innocuous than it is; notifications won't be ignored on the first or second day. Things will just slowly get progressively worse degree by tiny degree until everyone's pager is full of meaningless crap, no one notices real outages occurring, and the vendors arrive. You really need to stay on top of useless notifications. Hunt them down and eliminate them on a regular basis, ask people if they care about the notifications they're getting, see if you can get a policy setup that requires problem acknowledgments for every notification, etc.

I should also make the point that, in this context at least, you aren't special. It's tempting to believe that the correct number of notifications for your organization is a subjective thing and that you don't need to worry too much because your recipients are savvy. Let me be clear—I don't care if you're surrounded by the floating disembodied brains of particle physicists where you work inside the singularity beneath the LHC, or by coffee machines harboring nascent AIs over at JPL, they will hate you if you send them too many notifications, and "too many" is an integer that can be derived by a formula that returns the absolute number of notifications you should be sending given the number of hosts and recipients in your environment. My point is, this is a universal truth of human nature, and you NEED to worry about it; savvy is NOT an input variable. I'd give you the formula, but I haven't had a chance to work it out yet (when I do, I'm writing a LISA paper about it, though).

3. Eliminate Email Dependencies

Once you've minimized the number of notifications you send, you should proceed to make darn sure the ones you're sending are getting delivered. A few months back I wrote an entire article [3] on the subject of this tip, so I'll spare you the rant and summarize by saying that my faith in email is waning. Instead, I recommend text via SMS, voice notification via Asterisk, or a combination of the two. Email-SMS gateways are OK, a real SMS modem attached to the system is better, real SMS with voice backup is best. My article walks you through the configuration of all of that.

Even if you do stick with email, an out-of-band backup is a great way to make the monitoring system resilient against network outages, which is helpful for the same credibility-related reasons listed in tip #2.

4. Monitor the Monitoring System

This one is self-explanatory. Few of us are good at introspection, and Nagios being no exception, it's wise to have at least a couple of heartbeat scripts somewhere off the monitoring system to make sure the box and daemon

are running. In the past, we had separate boxes for monitoring and logging, with the logging box watching Nagios. These days I have multiple special-purpose Nagios systems watching each other.

5. Have a Naming Convention

I cannot stress enough how important it is to have a predictable naming convention for the hosts and services in your Nagios implementation. The CPU_LOAD service should be called the CPU_LOAD service everywhere it is consumed, measured, reported on, and referred to throughout your environment. It should transcend disparate monitoring systems, executive reports, event-correlated databases, and Web front-ends. The host called fooServer02.hq.com should be referred to everywhere as exactly fooServer02.hq.com, not fooServer2, or fooserver02.hq, or any other derivation thereof. "www.foo.com" should never be referred to as "fooweb" or "the foosite" or anything other than "www.foo.com."

Effective monitoring systems grow. They quickly become relied upon to prove SLA compliance and provide re-purposed availability information to executives, customers, and other technical staff members. When this happens, programs will be written to query and move data around. As things get bigger, ancillary systems will come in—RRDtool, Cacti, etc. If you aren't anal-retentive about names from the get-go, then things will quickly devolve into a kludgey mess. The RRDtool database referring to fooServer02.hq.com will not match the name in Nagios, or people will write scripts assuming different names. Data tables and reports will be empty for some systems but not others, and it will appear to be the monitoring systems' fault for not collecting data.

Worse, it's nearly impossible to fix these sorts of problems without a proper and agreed-upon naming convention in place. Every new system, service, or change introduces the possibility of another statically coded name exception in one or more of the four thousand tiny data-mover scripts. Credibility is quickly lost in an environment like this.

Your naming convention should be so pervasive that literal service names start leaking into human vernacular. When people start saying things like "CPU underscore Load" in meetings, you're on the right track.

6. Aggressively Collect Performance Data

You may have wondered in tip #2 why you would want to monitor lots of services if you weren't going to notify on them. Performance data is the answer. There is no good reason why you shouldn't collect performance data on every service that you poll. In Nagios even plugins that don't officially return performance data via the pipe syntax can be parsed directly for performance data. Tools are available to completely automate the detection of new services on new hosts and to create and maintain round robin databases of performance data for them. Even if you don't have the means or the inclination to display performance data, you should be collecting it in case you ever want to.

For years I have used the combination of NagiosGraph [4] and Drraw [5] to glue Nagios to RRDtool. NagiosGraph does an awesome job of completely automating the task of getting data out of Nagios and into round robin databases. It detects new services and hosts using regular expressions, and creates new RRDs as necessary. After the initial setup, you don't need to do a thing, and you'll have performance data for every service on every host you

monitor. Drraw is a super-simple CGI-based Web app that takes a directory of RRDs and gives you interfaces to draw anything from individual graphs to dashboards. It is the most flexible interface for drawing graphs from RRDs I've used. It makes it easy for me to quickly draw a graph that cross-references data from all sorts of hosts in different locations, and I don't feel I need to keep the graph around. Usually I just draw it to get a question answered and never save it at all. This sort of quick, informal graphical troubleshooting has become an important tool for me, and I'd be an unhappy sysadmin without it. I highly recommend both of these tools.

7. Implement Purpose-ful Nagios Systems

Finally, if you have the resources, it's a great idea to consider running disparate Nagios daemons for different purposes. For example, we run two different kinds of Nagios daemons where I work, "internal" and "external." The internal Nagios hosts sit in the production environment with the production systems and query the NRPE-type services: CPU, memory, swap, disk space, ps lists, and the like. The external Nagios daemons sit on the public Internet and act like customers, logging into the public Web sites, authenticating, clicking around, doing things that humans do.

We don't bother rolling up alerts, preferring instead to let Nagios hosts individually contact us about things they think are wrong. In this way we get a much better feel for not only how reliable our services are but how reliable our Nagios hosts are, and we gain a measure of clarity about a given problem based on which hosts are complaining and about what things. A box on an XO link in California complaining about a problem that hosts in Texas and Pennsylvania don't see could imply an upstream network outage, for example. This also has a tendency to keep the server configuration simple and transparent.

That about wraps it up. I hope these weren't overly obvious and that you perhaps found something that might help you out in the future. There are a lot of places to easily go wrong implementing monitoring systems, so oftentimes it's the human equation that makes a huge difference between good systems and bad ones. A huge difference, I dare say, between two sides of a very fine line.

Take it easy.

REFERENCES

[1] djbdns: http://cr.yp.to/djbdns.html.

[2] Holt-Winters and exponential smoothing: http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc437.htm.

[3] Dave Josephsen, "iVoyeur: Message in a Bottle—Replacing Email Warnings with SMS": http://www.usenix.org/publications/login/2009-02/pdfs/josephsen.pdf.

[4] NagiosGraph: http://sourceforge.net/projects/nagiosgraph/develop.

[5] Drraw: http://web.taranis.org/drraw/.