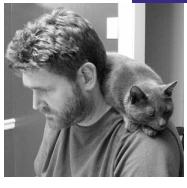
DAVID JOSEPHSEN

iVoyeur: a view from someplace nearby



David 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

GREETINGS. WELCOME TO *;LOGIN:'S* shiny new monitoring column. When Rik first approached me with the idea, I must admit my first thought was to wonder if there was enough subject matter to fill a semimonthly column for a reasonable length of time. Is systems monitoring really that deep? If you have any experience with the large enterprise-strength monitoring apps, then you know that vendors don't seem to think so; they view systems monitoring as a largely turnkey affair: Purchase license, install agent, reboot server, repeat.

Even the corporate-backed open source upstarts seem to share this opinion to a certain degree [1]. While the Patrols and OpenViews of the world clamor to support the largest number of gadgets, the Hyperics and Zenosses appear to be differentiating themselves based on their auto-discovery tools and ease of configuration. If the vendor claims of "zero to monitoring solution in 30 minutes" are to be believed, then a monitoring column might not be a particularly entertaining prospect for you.

But as a good friend of mine once (quite rightly) said, "Knowing that there is a Web server on port 8080 is about 2% of the problem." Systems monitoring, it turns out, is anything but a turnkey affair. Just behind the shiny facade of port scanners and SNMP traps lies a stunningly complex problem, involving a question, the answer to which is unique for each person who asks it. It is a problem in fact that I think we have yet as a community to fully understand, much less actually solve.

Consider for a moment what happens when you type a URL in your browser and get back an error page. At that moment, the actual status of the Web "service" in question is a quantum superposition; it is, to you, in a Schroedinger's state. You have observed an error page, but that isn't necessarily indicative of a problem with the Web site itself. There are a great many things that could be wrong that have nothing whatsoever to do with the Web server. The blame might rest with your system's network connection, DNS, an unfriendly filter, or a mistyped "ip route" command by some sleepy admin somewhere in the worldwide mass of interconnected routers between you and the Web page you seek. Some of these you can test for, and some are more difficult to detect. The Web site is up *and* it is

down. There is an objective reality—a singular state—but for the moment it eludes you. You'll have to tease it out.

Teasing things out, however, is a talent your monitoring system doesn't possess. It checks exactly the parameters you tell it to check, and it returns the result. If you called the parameter "Web service," then that's what the monitoring system will tell you is down, and if you aren't careful about choosing the parameters, it might even tell you everything is fine in the presence of a problem—an even more distressing proposition. If only knowing the state of the cat were as simple as opening the box. Arguably, the pinnacle of our error detection capability at this point is end-to-end monitoring, involving scripts that mimic user behavior, thereby encountering the same problems a user would. But end-to-end monitoring programs are somewhat of a copout, because they don't actually give you the state of the cat either. They tell you that there is a problem (from the perspective of the monitoring system), but not where the problem might actually reside. Their real intent is to catch errors that more specific checks such as port scanners might not. Monitoring systems, it seems, are not (yet) capable of making the observations necessary to solve our quantum conundrum.

So you can call this notification from your monitoring system a "Web outage" on the reporting interface if you like, but that doesn't make it true. Like the demanding helplessness of the user crying, "The interwebs are broken," there's information there, but not very much, and it's of questionable accuracy. Perhaps knowing where the problem lay is not critical to you; it's enough to know that there is a problem, and you'll take it from there. But perhaps automated site-to-site failover depends on bulletproof detection of a specific error or set of errors, or maybe the problem is chronic and requires a human to detect patterns in the service availability over time (false alarms make pattern hunting a bit more difficult). Either way, the monitoring system probably hasn't actually answered the question it was intended to answer, and many of the humans using the system won't be aware of the distinction. In systems monitoring, the area where the humans and system meet is especially problematic.

Really the core of the monitoring problem is that we've created ourselves some rather untrustworthy machines. There's just an awful lot of places where things can go bad, and for all of our fancy packet pitching, today's PCs are very much islands unto themselves, barely aware of their own state, much less that of the network around them. We, like an unfortunate mix between detective and geologist, rely mostly on forensics to gain what insights we can, using netflow, syslog, utilization graphs, and monitoring tools. And being every bit as untrustworthy as the systems they are trying to monitor, the monitoring box itself can have all of the same problems. In the end all it can give you is its own crudely gleaned opinion of the current state of a set of services from a single static point in the network, which is often a poor substitute for knowing the service state firsthand.

So asking one fallible machine in a fallible network its opinion about the fallible machines surrounding it might not be so great an idea. Doing so is not unlike paying a guy $5 to watch your car at 2:30 a.m. in Tijuana. (Usually it works out fine, but that doesn't make it a good idea.) And speaking of misplaced faith in humanity, the humans in this equation are equally as fallible as the machines (if not more so). For one thing, in classic, failure-to-quantify-the-risk fashion, we sysadmins and our managers seem to place an unfounded amount of trust in our monitoring systems. As if calling them "monitoring systems" somehow imbues them with a magical immunity from mistaking a DNS failure for a Web site outage, or even just crashing outright.

But alas, our monitoring tools betray us. They crash like normal systems and are largely dependent upon the same network infrastructure as the other systems. And yet for some reason the false positives surprise us as much as false negatives; it "feels" like this sort of thing shouldn't happen to the monitoring system. It seems ironic, when there's no real reason it should. It's telling even that I used the word "betray." So there is an emotional component here, and its most common effect is to cause us to ignore a monitoring system that has proven itself to be unduly chatty, or sometimes incorrect. We don't "lose faith" in Tomcat when it runs out of threads and starts handing out 500s, but for some reason we are quick to anthropomorphize and discredit a monitoring server for its digressions, even though it may be the worst possible server to take with a grain of salt.

With "normal" systems—the ones without the magical "monitoring system" moniker—we mitigate the risk of failure with redundancy, incorporating load balancers, VRRP, and BGP multihoming; redundancy is an industry unto itself, and it could certainly help out in a monitoring context. It's not uncommon for a large organization to have a failover monitoring box, and large installs sometimes require numerous monitoring systems to aggregate alerts to a master in order to scale, but these setups don't improve the resolution of our failure detection ability.

Parallel systems have potential in this regard; two opinions are better than one. Yet curiously, monitoring systems are seldom deployed this way. (If you have one, I'd like to hear about it.) This might be because having parallel monitoring systems agree on a given service state is a difficult problem to solve, which in itself is a decent proof of the fallibility I just alluded to. What do you do when two systems disagree? Further, avoiding things such as redundant notifications requires that the monitoring systems be somewhat aware of each other's opinion of the current state of things, making prospects even hairier.

Leslie Lamport is intimately familiar with getting parallel computational entities to agree on states. His work on the Byzantine Generals [2] problem is used widely at NASA and the aerospace industry to design fault-tolerant flight-control systems. His work, and the work of those at SRI, showed that $3n + 1$ processors are generally required to tolerate n faults. In layman's terms and warped to suit our needs, the opinions of 4 monitoring systems would be needed to reach a trustworthy agreement on a given service if one of them were malfunctioning. I don't have any fancy math to back this up, but it "feels" like the odds of 1 out of 4 PC-based monitoring systems misbehaving at any given moment are good. So monitoring systems, it seems, may need to be a bit more redundant than we're used to before they can begin to give really meaningful opinions. We can't simply toss another box in without making things a lot more complicated, and it turns out we'd need to throw in quite a few before seeing a real return on the investment.

None of this is to say that systems monitoring is impossible or hopelessly broken. In practice monitoring tools usually work pretty darn well, and they are certainly better than nothing at all. My monitoring systems have saved my gravy more times than I can remember. But it's useful, I think, to imagine a reference system, some inexpensive, Byzantine failure–proof, massively parallel monitoring system communicating securely via out-of-band channels and telling us with flawless accuracy and resolution about specific problems and their causes without burdening the network with traffic or the systems with bulky agents. It introduces no security flaws, has an infinite amount of trending and utilization data on every metric we can imagine on every server, has a network device in the environment, and can do complex event correlation and aberrant behavior detection in real time. Maybe it has

some of those heuristics and biological diversity I'm always reading about, and what the heck, it runs Plan9, and doubles as a margarita machine. This makes it easier to imagine the huge space of gray between the reference system and the system you probably have in your shop today. That enormous gray space is what the vendors are ignoring when they say, "0 to monitoring solution in 30 minutes."

So, needless to say, I happily took Rik up on his offer. In this column, I want to explore the gray space, providing practical solutions, advice, code, and general food for thought. My sincere hope is that perhaps somewhere along the way we'll both gain a better understanding of the problem, and maybe move a few gradients closer to the monitoring system of our dreams. Expect topics to range from network architecture to SNMP to security to data visualization to temperature sensors to dealing with humans and back again, running the gamut of what you as a sysadmin might run into in the course of implementing and maintaining a monitoring system.

To a large extent the information I provide will be specific to Nagios [3], which is probably the most nearly ubiquitous open source monitoring program today. This is not, however, a column about Nagios. I would prefer that you think of Nagios as a reference implementation language rather than as a design requirement. If systems monitoring has an XML-like means of specifying solutions, a prototyping language that is relatively easily translated between disparate systems, then Nagios, with its (almost painfully) open architecture and liberal lack of design assumptions, is probably the closest thing I've seen to it. So my use of Nagios in this column is only to ensure that the solutions discussed herein have a good chance of being translated to whatever you happen to use (and if that's Nagios, then all the better).

Feel free to shoot me email [4] or comment on my blog [5] if you would like to talk about something specific or just want to say hi. And finally, believe it or not, I honestly plan to maintain a better signal-to-noise ratio in my future articles, so sorry for the theoretical ramble. I promise to have some nitty-gritty for you in the next issue. Take it easy.

REFERENCES

[1] http://books.slashdot.org/comments.pl?sid=230333&cid=18695063.

[2] http://research.microsoft.com/users/lamport/pubs/pubs.html#byz.

[3] http://www.nagios.org.

[4] dave-usenix@skeptech.org.

[5] http://www.skeptech.org.