iVoyeur

All Together Now

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

I attend conferences often enough that they sometimes melt together in my memory. It's for this reason, I think, that I can't be sure where I was first introduced to what's become known as the "Google Commodity Server Model." It was a Google talk at some USENIX conference, I'm sure, and had to have been before the demise of Compaq as a server company, because that was the image it destroyed in my mind. I'd come expecting to be dazzled with a dream-world of orderly, sterile white-space. It would be thoughtfully arranged with row after row of gleaming-white racks housing 8 and 16u beige Compaq behemoths as far as the eye could see. Brainy grad students moonlighting from Stanford or Berkeley would loiter in white lab coats and consult their clipboards while thoughtfully discussing the output from the hundreds of surface-mounted displays in the NASA-like control-center of a NOC.

Perhaps you imagined something similar. I find it comforting to believe I wasn't alone. Maybe you were even in the room with me. There we sat having our expectations smashed by photos of Google's dimly lit cement-floored warehouses. Their doorless black racks literally bulged with sagging desktop motherboards haphazardly shoved into place atop sheets of plexiglass (plexiglass!), their hard drives and RAM shamelessly exposed to the world. Like lumbering, unhappy mules, grungy liberal-arts majors from the nearby community college trudged down the stone passageways in their sweat-stained Creed T-shirts, heaving before them shopping carts (shopping carts!) laden to the brim with replacement hard-drives and DIMMS. We couldn't have been more wrong. Really, what *were* we thinking?

But then, who could blame us? For that time, and for that industry, the model was almost satirical in its absurdity. And like witnessing great satire, we were at first horrified, and then fascinated, and ultimately convinced. One quickly realized that they had a point—either stumbled upon out of necessity, or the proof of a powerfully well-devised hypothesis. Google had at once posited and proven that if you took enough hardware and used it to build a singular, parallel application, then it didn't matter how cheap the hardware was or whether it broke on a regular basis. There are many reasons commodity clusters are triumphing over supercomputers and other large production systems, but this, I think, is one of the most compelling: it's easier (and cheaper) to manage production systems when we can accept as a matter of course that some subset of it is going to be broken, because that is the practical reality, no matter what the scale of infrastructure in question.

Google changed the sysadmin game from large, reliable machines with failover to parallel active clusters of smaller computers and, as a result, never had to worry about system reliability, or vendor lock-in, or the financial viability of one vendor vs. another, or a 24/7 NOC staff, or uninterruptible power, or a hundred other frustrating nits that we all fought management over in the '90s (although many IT organizations still haven't quite caught on). This is arguably one of the minor ways in which Google has changed the world, I suppose, but it was an important one to the sysadmins among us, and while very few of us have plexiglass in our racks, it's a rare thing, I think, to find a Web site of any size that doesn't have an active parallel cluster of small systems behind it (even if those systems hide their components behind a Dell-branded faceplate).

A few weeks ago I was at lunch with some friends who still work at my last place of employ, and they mentioned one of their sites being down as a result of an IIS update gone wrong. I was actually confused. The Web site was down because a server was down? How does *that* work? At this point there is in my mind no singular, straight line that connects the application to the server hardware, and that is a telling transposition. The application is a bulbous, free-floating organism. Its appendages overlap virtual machines, penetrate network subnets, and transcend physical hardware. For me at least, and I suspect much of the LISA crowd, putting a Web site on a box is a design that no longer computes. It simply isn't how we build infrastructure support for Web applications anymore.

This being a monitoring column, the question begged by my typically wordy preamble is: given that clusters appear to be changing how we build application infrastructure, are they also changing the way we monitor application infrastructure and for that matter, how we monitor the applications themselves? The answer is of course "Yes," and also "duh," but at the risk of telling you what you probably already know, I thought I'd spend this month's column exploring some of the ways clustering is, for me at least, changing systems monitoring.

In the production environments I build and maintain, everything is pretty much a cluster up to and including the routers and firewalls (which are OpenBSD systems), as well as the load balancers (which are fancy Apache boxes). Our monitoring efforts are numerous and wide-ranging, but here I'd like to focus on availability monitoring of clustered applications. In this context I believe there are a few categories that are directly affected by a clustered infrastructure, for better or worse.

The first is the use of an external end-to-end application check. By this I mean a system or set of systems, located outside your network, that monitors the application by using it the same way a normal human would. Given that static content and dynamic content often reside on different systems in different networks, and the myriad, subtle ways application servers tend to fail, it is not a trivial undertaking to define a Web-application outage, much less detect one. The site might be up, for example, but not displaying graphics, or it might be up except running out of threads every so often, and so on. When you add a cluster of application servers to this already complex problem, you're adding all sorts of new failure models. For instance, one server out of the 20 might have an unstable database connection pool, or the application content might not be properly synchronized across all cluster nodes.

If external end-to-end application monitoring wasn't a requirement before clustering, it certainly is the only way to be sure a given application is running well in a clustered world. If your developers follow a formal software development life-cycle, chances are a test plan for the application gets created at some point. I recommend taking that test plan and transforming it into an end-to-end monitoring check. We use an internally developed framework based on curl and run these checks from a Nagios box hosted in an off-site colo. Sometimes it's sufficient to visit the home page and verify that a few links are present. Other times we execute logins with test accounts and follow several links into the app. Pay particular attention to how the application manages sessions, especially if they are sticky; the monitoring system needs to be able to traverse different paths through the cluster rather than running its check against the same cluster node over and over again.

In our environment we usually find it sufficient to simply dump our cookies in the bit-bucket after every check. Designers of large clusters should put some thought into designing a session-state system that allows the monitoring system to specify its own path through the balancer tier. It may be necessary to use multiple external monitoring nodes for very large clusters (more on this later). In the context of external, end-to-end monitoring, the introduction of clustered systems has complicated things.

Another way in which the concept of commodity clustering has complicated the task of monitoring is by introducing an entirely new category of systems monitoring—that of automated node management. Now that we have 4 or 30 or 800 cluster nodes, we need to automatically fail them out of the cluster when they fail. Depending on the type of cluster involved and what failure means, this can be a serious headache. For routers and load balancers, it's usually sufficient that each node is responding on a common IP. Various multicast MAC approaches such as CARP, HSRP, or VRRP are normally used here. When the cluster nodes are running a more complex application, things get hairier, but whatever the details of the nodes themselves, we can't use a traditional monitoring system like Nagios here.

In the first place, we need to detect failed nodes quickly, at the most within a few seconds of the problem occurring. Secondly, the detection model needs to be peer-to-peer. It's a bad idea to depend on a single point of failure, like a management and monitoring node, to tell us when a cluster node is up or down. Third, this monitoring layer needs to take immediate action to offline the node when a problem occurs. This introduces a slew of new problems such as having a fail-safe mechanism for offlining a node, what constitutes a sane timeout, whether or not to automatically re-enable a node, and capacity planning concerns such as whether the load from the failed node will balance gracefully across the rest of the cluster and whether the rest of the cluster will be able to bear the brunt of losing the node.

We've had to solve these problems several times over in our production environments for different types of clusters, and despite several attempts to bring commercial hardware and software solutions to bear, we always seem to return to assembling our own tools from the same small assortment of open source programs. We've had good experience with CLUSTERIP [1] in the Linux kernel, as well as OpenBSD's CARP [2] and UCARP [3] on Linux. Apache's mod-proxy-balancer [4] does a good job of quickly detecting and disabling app-tier nodes that are not responding on their respective ports, and it has a nicely automatable node management interface. Finally, we've written a fairly complex application-layer node management framework that we refer to internally as "webstated" to detect higher-order application server errors and disable nodes accordingly.

Automated cluster node management is, in my experience, a hairy endeavor with simple applications. It is made nearly impossible (or at least maddeningly frustrating) with cluster nodes that are running unreliable and strangely behaving Web-application frameworks. Our various solutions are stable and I'm quite happy with them, but given the amount of attention we've had to pay to this problem, I'm often surprised by the simplicity of the commercial offerings in this context. I suspect monitoring for node management is far from being a solved problem in the world of commercial balancers, not to mention that the commercial balancers themselves cannot usually be actively clustered.

Speaking of systems that aren't actively clustered, I've long been fascinated by the idea of having a cooperative cluster of monitoring systems. Monitoring clusters to date have focused on scalability to meet the demands of monitoring large infrastructures, but a small patch to something like the DNX [5] framework could give us a group of monitoring servers that all monitored the same small group of services, such as a series of external Web sites. Instead of increasing the possible checks per second, this could increase the quality of the monitoring result. A cluster of monitoring systems could watch the same service from different networks and cooperate to form an opinion about the state of the service in question. This could rectify all sorts of false alarms, accounting for failures in individual monitoring nodes and DNS and network provider issues.

There are several commercial entities providing this sort of service ([6], [7]) but if your Web site itself consists of several different clusters of hosts—for example, a balancer cluster, an app-tier cluster, and a database cluster—then you're going to need some flexibility in your monitoring cluster, for the reasons I've mentioned. A monitoring cluster does you no good if all the monitoring nodes are hitting the same app-tier node, or if they're all hitting different app-tier nodes but you can't tell which. To be sure, clusters generally complicate our monitoring endeavors.

In one way at least, Google's commodity cluster model has made life easier in a monitoring context. Once we can be sure that the application is in fact functional and that failed nodes can be reliably detected and disabled, we need not care as much about the traditional alerts we associate with systems monitoring. I, for example, no longer receive alerts from Nagios that a given service on a given host is down, or even that the entire host itself is down. Instead, I receive a warning when a given cluster has degraded to 66% capacity, and that's the sort of thing that can wait until morning.

Take it easy.

References

[1] ClusterIP, cluster support in Iptables: http://security.maruhn.com/iptables-tutorial/x8906.html.

[2] Common Address Resolution Protocol: http://www.openbsd.org/faq/pf/carp.html.

[3] User-space CARP: http://www.ucarp.org/project/ucarp.

[4] Mod_proxy_balancer: http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html.

[5] Distributed Nagios Executor: http://dnx.sourceforge.net/.

[6] BrowserMob clustered systems monitoring: http://browsermob.com/website-monitoring .

[7] Keynote clustered systems monitoring: http://www.keynote.com/.