iVoyeur
The Gift of Fire

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

It was Prometheus, so the story goes, who gave us fire. The other titans being otherwise engaged, it fell to him and his equally guilty but rarely talked about brother Epimetheus to populate the earth with flora and fauna, or at least to furnish them with the good stuff. Being neither solid architects nor effective planners, they may not have been the best choice for the job, for by the time they got around to the humans, they'd completely run out of horns, and antlers, and fangs, and shells, and wings and whatnot.

One imagines them standing wide-eyed over their empty basket of measures and countermeasures, like a spent bucket of Legos with the tower half-built, each accusing the other of some fatal error in accounting. "Gah," says Prometheus, being the bigger man [1] of the two. "I'll take care of it," and off he stomps into the heavens, eventually returning with fire and, bestowing it upon us (perhaps along with some fingernails that happened to be lying unwanted in the bottom of the basket), he calls it a day.

It was, for him, a fatal gift. A gift for which, as you may have heard, he was chained to a rock for eternity to have his entrails repeatedly devoured by a vulture—being divine entrails, they always grew back. That punishment always puzzled me. I mean being chained to a rock for eternity already seems like overkill, but the entrails thing, that smacks of a virulent and concentrated malice. Some ancient analog of whatever it is that makes a person reload the gun so that they can continue shooting bullets into the already dead body of their spouse, or boss, or spouse's lawyer, or boss's lawyer, or really any lawyer. And for what? Giving us fire? It seems like we would have probably figured that out on our own eventually. Why so severe a penance?

I suspect it was not the fire itself, which we ourselves could re-gift to a rhinoceros tomorrow with no substantial effect (save perhaps what would normally be expected from a freaked-out rhinoceros), but, rather, the propensity to use that gift that got him in trouble. The gift of fire is not the same thing as the gift of being able to use fire. Quite a bit more is implied by the latter. Perhaps the word "fire" is meant metaphorically, and what we were truly given that day was an inner-fire; that indescribable self-awareness that makes us human. The flame in us that not only enables us to use fire, but allows us to think about fire, and what we think about when we think about fire—now *there's* a gift over which I can imagine someone wanting to sic a vulture on a dude.

Awakened Rhinoceros

Over the course of theplast few months I've been having what can only be described as conversations with my Nagios servers. Admittedly, these conversations are entirely composed of bits of information that I could get from the Nagios Web Interface, but the relevant point is that questions are not only being asked but answered. There is a dialog. Where before there was a black box, a wholly self-contained system comprised of a daemon that spoke to its own UI and nothing else, there is now an introspective entity to which I can posit queries, and from which I can expect replies. Before there was a rhinoceros and now there is, well, it's still a rhinoceros, but now it can tell me how it's feeling, which I think you'll agree is a cool trick for a rhinoceros.

Have I finally gotten with the program and installed NDOUtils [2] on my Nagios servers, pushing the state into a MySQL database so that I can query the DB? No. I've avoided writing an article on NDOUtils for the simple reason that I don't use it. It complicates things, doesn't scale well, and encourages a mindset I don't think works very well in distributed monitoring architectures, specifically, that all UI problems are nails and PHP/MySQL is the hammer. But that's another article.

I'm talking about MK Livestatus, the second of the three gems I plan to cover written by Mathias Kettner, a man who should be wary of rocks and vultures, because Livestatus does for Nagios something very similar to what Prometheus did for you and me, imbuing Nagios with, if not the flame of self-awareness, then at least the spark of introspection. If you've ever tried to script against a Nagios system, you understand the problem all too well. There just aren't a lot of ways to get data out of it, the three most popular being to scrape the Web interface, to parse status.dat, or to use NDOUtils. Over the years I've covered a few add-ons to assist in this endeavor, including my own beloved filesystem event-broker module, which exports the Nagios state to a file system.

Livestatus has all of these beat. Instead of replicating the state data to a log, or a database, or even a file system, it simply provides you with a UNIX socket (not a network socket) through which you can query state information directly from the Nagios daemon itself. You ask questions via a simple query language, and receive immediate answers directly from the same data structures resident in memory that are in use by the Nagios daemon. There is no overhead associated with replicating state or querying external databases. There is no disk I/O whatsoever. There is no latency and no risk of the data being out of date, and, importantly, there is no risk of blocking the daemon process with a hung ODBC thread or something similar.

MK Livestatus is implemented as a Nagios Event Broker (NEB) module, which is a compiled, shared-object file that is loaded into Nagios at startup. NEB modules communicate with the Nagios daemon by subscribing to announcements made by the Event Broker, and generally have access to everything that the daemon itself has to offer. I covered the Event Broker and the creation of NEB modules in depth in [3], and there are now several good articles in the tubeosphere, including official documentation at [4].

Livestatus is included in the Check_MK tarball, but can also be downloaded as a stand-alone package. If you're installing Check_MK, which we covered in last month's article, simply answer "yes" when the setup script asks if you'd like Livestatus installed. The stand-alone build is the autoconf standard configure, make, and make install. Either way, Livestatus.o will be compiled and placed in /usr/lib/check_mk by default. Edit the nagios.cfg to enable the event broker with:

```
event_broker_options=-1
```

… and tell Nagios to load Livestatus with:

```
broker_module=/usr/local/lib/mk-livestatus/livestatus.o /var/lib/nagios/rw/live
```

Restart Nagios, and you're all set. The second argument is the location of the Livestatus socket on the local file system. Again, this is a UNIX IPC socket, so you'll need a tool like socat [5] or unixcat [6] to work with it in shell. Livestatus actually comes with its own copy of unixcat, which it will install for you in /usr/local/bin by default. I'm unsure whether this is the same tool as the one in ucspi-unix or if it's original code. Caveat emptor. The suggested method for making remote queries to Livestatus is by piping them over SSH, but there's no reason why the UNIX socket couldn't be front-ended by a TCP server program such as xinetd or DJB's tcpserver. Sample configuration for xinetd is available at [7].

Several other options may be passed to the NEB module on the broker_module line to, for example, place it in debug mode, or tune the various cache, buffer, and thread sizes. These are fully documented at [7]. Once Nagios is restarted, check to make sure the socket exists in the expected location, and test that it's working with:

```
echo 'GET columns' | /usr/local/bin/unixcat /var/lib/nagios/rw/live
```

If everything has gone according to plan, a dizzying block of text should scroll by. The "GET" statement in the above command was an "LQL" or Livestatus Query Language command. LQL commands are made up of two parts, the first line, which always begins with "GET", identifies the object type you're interested in inspecting. The Livestatus documentation refers to these as "tables," but they roughly correspond to object types in Nagios. The available tables, shamelessly pasted from the official documentation at [7], are:

**hosts - your Nagios hosts*
**services - your Nagios services, joined with all data from hosts*
**hostgroups - you Nagios hostgroups*
**servicegroups - you Nagios servicegroups*
**contactgroups - you Nagios contact groups*
**servicesbygroup - all services grouped by service groups*
**servicesbyhostgroup - all services grouped by host groups*
**hostsbygroup - all hosts group by host groups*
**contacts - your Nagios contacts*
**commands - your defined Nagios commands*
**timeperiods - time period definitions (currently only name and alias)*
**downtimes - all scheduled host and service downtimes, joined with data from hosts and services.*
**comments - all host and service comments*
**log - a transparent access to the nagios logfiles (include archived ones)*
**status - general performance and status information. This table contains exactly one dataset.*
**columns - a complete list of all tables and columns available via Livestatus, including descriptions!*

As you can see, these are mostly Nagios object types. You may have noticed "logs" in the list. Your eyes do not deceive you; Livestatus can retrieve log messages for you, even if they've been archived. It does this by subscribing to log message events from the event broker and then caching them in memory. The number of cached messages in this memory buffer is one of the options you may pass in on the broker_module line. The default value is 500,000.

The second part of an LQL command is made up of any number of subsequent lines, collectively referred to as "headers." Headers generally allow you to modify your query, restricting the output to a given set of columns, filtering it for objects that meet a criterion, transforming the output into a statistic (e.g., returning how many objects matched, or computing the average of a returned value from several hosts), and even modifying the output format (to JSON vs. CSV, for example).

I used `GET columns` as the initial test command because a command like `GET hosts` will return a massive amount of data from a moderately sized Nagios server. Now, with the help of the "Columns" header, we can return just the host attributes we're interested in, like so:

```
Q='GET hosts
Columns: address state
' echo "${Q}" | unixcat /var/lib/nagios/rw/live
'
```

The "Columns" header causes Livestatus to return only the named attributes. The query above should return a list of the address of every host monitored by Nagios, followed by its current state. The returned list will use semicolons for line separators. Because LQL queries are themselves multi-line, things get a little weird from the command line. In the above example, I'm setting a variable to the multi-line query and then echoing it to unixcat (the double quotes around the variable in the echo statement are important. Without them the shell will eat the line-feeds in the variable). The official documentation recommends that you save the query to a file, and then redirect it to unixcat, like so:

```
echo 'GET hosts
Columns: address state' > query.lql

unixcat /var/lib/nagios/rw/live < query.lql
```

I quickly tired of both of these methods and took a minute to write a small shell wrapper that I call "qls" (query Livestatus) to make it easy to interact with Livestatus. It works kind of like an interactive Livestatus shell and is available for download from the USENIX *;login:* site at [8].

If I wanted to narrow my list to just the hosts that were in a critical state, I could modify my query accordingly:

```
GET hosts
Columns: address state
Filter: state = 2
```

I can continue to narrow and modify my results with as many headers as I'd like, placing each new header on a subsequent line. The "Filter" header is quite powerful, supporting 12 equality operators, including negation, and case sensitive and insensitive POSIX extended regular expressions ("~" for case sensitive, and "~~" for case insensitive). I could, for example, refine my search to all the hosts whose description began with "DB" followed by a number between 1 and 4 (inclusive) with:

```
GET hosts
Columns: address description state
Filter: state = 2
Filter: description ~ ^DB[1-4]
```

Filter headers can be combined by suffixing them with a logical AND or OR header. This makes it possible to, for example, return the DB hosts that are in either a critical or warning state, like so:

```
GET hosts
Columns: address description state
Filter: state = 1
Filter: state = 2
OR : 2
Filter: description ~ ^DB[1-4]
```

The ordinal after the OR header defines how many of the previous headers are included in the logical OR.

Sometimes you don't want a list of the actual things, but, rather, the number of things that meet your criteria. The "Stats" header can not only provide counts but also compute sums, averages, and standard deviation, as well as return Min and Max, and much more. The following query returns the number of hosts in a critical state:

```
GET hosts
Stats: state=2
```

Stats headers also support logical operators. Here is the number of hosts that are in a critical state whose state has not been acknowledged:

```
GET hosts
Stats: state=2
Stats: acknowledged=0
StatsAnd: 2
```

The last example I have room to give is of the "Stats" headers grouping support. By adding a "Columns" header to my Stats query, I tell Livestatus to output a line each time the given Column value is different. For example, the following query will break down the number of unacknowledged critical hosts per hostgroup:

```
GET hosts
Stats: state=2
Stats: acknowledged=0
StatsAnd: 2
Columns: groups
```

I'm barely scratching the surface here, so if any of this was the slightest bit interesting to you, I'd highly recommend checking out [7]. Mathias Kettner really has knocked it out of the park with Livestatus. It's probably the coolest Nagios add-on I've encountered. It's so great it should just be in nagios-core. Next time we'll take a look at Multisite, a new Web GUI for Nagios which uses Livestatus as its datasource.

Take it easy.

References

[1] “The Bigger Man,” American slang for the first person to back down from a conflict. http://www.urbandictionary.com/define.php?term=Bigger%20man.

[2] NDOUtils: http://exchange.nagios.org/directory/Addons/Database-Backends/NDOUtils/details.

[3] Dave Josephsen, “*You* Should Write an NEB Module,” *;login:*, vol. 33, no. 5, Oct. 2008: https://www.usenix.org/publications/login/october-2008-volume-33-number-5/ivoyeur-you-should-write-neb-module.

[4] NEB API: http://nagios.sourceforge.net/download/contrib/documentation/misc/NEB%202x%20Module%20API.pdf.

[5] socat, the multipurpose relay: http://www.dest-unreach.org/socat/.

[6] ucspi-unix: http://untroubled.org/ucspi-unix/.

[7] Livestatus docs: http://mathias-kettner.de/checkmk_livestatus.html.

[8] www.usenix.org/publications/login/august-2012/ivoyeur.