iVoyeur

Hearsay Among Monitoring Systems

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

This may be a little premature to talk about, but lately I've been consumed by an idea that is conceptually rooted in the complexity involved in making monitoring systems talk to each other. For someone who writes articles about making monitoring systems talk to each other, this is perhaps natural, but I know I'm not the only one who has noticed that adding a new monitoring system to an existing infrastructure does not linearly increase its complexity.

For example, say you have Nagios and want to add Splunk, and you want them to talk to each other, feeding passive check results from Splunk to Nagios and also round-trip times for an HTTP service in Nagios into Splunk. Then you add Ganglia and Collectd to the mix in a similar fashion. This scenario, depicted in Figure 1, begets four custom configurations for Nagios alone, one for Nagios itself, one for Nagios to talk to Splunk, another for Nagios to talk to Collectd, and yet another for Nagios to talk to Ganglia. Some of these systems will need to be configured in kind to talk back to Nagios.

I/O Hooks Aren't Enough Anymore

So inter-system configuration complexity is something like $(n\text{-}x)^2+(nx)$, where x is the number of send or receive-only, Graphite/Collectd-style tools you plug in to your monitoring architecture. If we were talking algorithms, we'd reduce this to $0(n^2)$ and be done. Effective systems monitoring requires a toolbox, but every tool you add to the box means reconfiguring all tools.

This complexity is obviously a hassle, but worse, it has a tendency to make snowflakes of your monitoring systems, eventually resulting in highly customized, fragile infrastructure. The alternative is to limit our visibility by forgoing the use of good tools to avoid the configuration burden (or installing them as stand-alone). A nearly exponential increase in configuration complexity makes this a hard limit for everyone, which is to say every shop WILL have to pick and choose a few tools from an increasingly huge list of amazingly great monitoring systems if they want them to work together.

At the risk of sounding melodramatic, I am saddened by this. I want all of these great monitoring tools to work like Legos. I want to plug them in to each other and build things with them. I want them to play to each other's strengths and become more than the sum of their parts.

There's No I in "Common Data Model"

Imagine for a moment that instead of each system having its own unique I/O hooks, they all supported a common data interchange format. If they all just woke up one morning and agreed to send and receive the same format messages. As depicted in Figure 2, they would no longer need to be configured specifically to communicate to each other, and could instead each be configured simply to enable import and/or export of the common format. Each

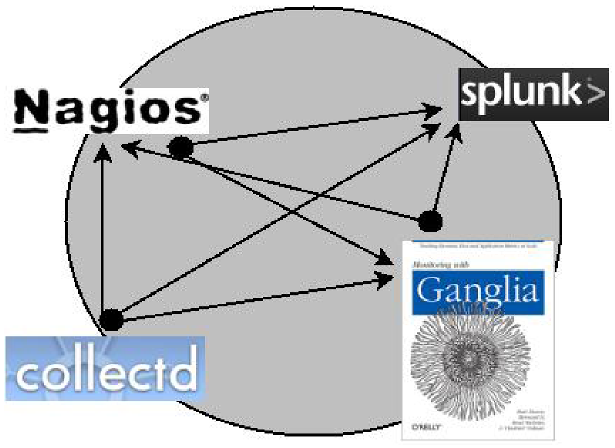

Figure 1: Each system must be custom configured for interoperability with the others.

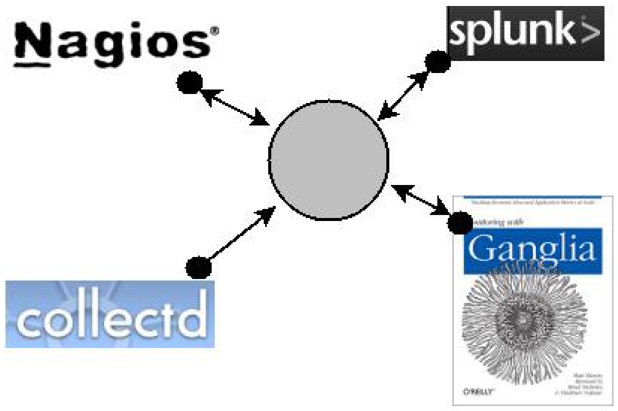

Figure 2: Each system merely enables support for a common data model.

monitoring system could share data out in a system-agnostic way, and other systems could pick and choose the state and metric data that was relevant to them regardless of the source.

This would greatly reduce the cost of adding new monitoring infrastructure, and would make everyone's life easier. But is it even possible to translate the output of every monitoring system to a common format that works as the input of every other?

Although it seems unlikely, practically speaking, all monitoring systems deal with similar data. The Riemann Project's event type, described at [1], characterizes a system-agnostic blob of monitoring data pretty perfectly. Copied directly from that site, the structure looks like this:

host	A hostname, e.g., "api1", "foo.com"
service	e.g., "API port 8000 reqs/sec"
state	Any string less than 255 bytes, e.g., "ok", "warning", "critical"
time	The time of the event, in UNIX epoch seconds
description	Freeform text
tags	Freeform list of strings, e.g., ["rate", "fooproduct", "transient"]
metric	A number associated with this event, e.g., the number of reqs/sec.
ttl	A floating-point time, in seconds the event is valid for

Every monitoring system I've worked with generates data that fits pretty well into this struct, and most fit with room to spare. Formalizing this, changing the "state" field to a Nagios-style int, and adding a UID field to make it possible to sign the messages and/or provide a unique hash so that they can be more easily de-duplicated/commuted etc. produces my own definitions:

string	Host //hostname, e.g., "foo.com",
string	Service //e.g., "HTTP reqs/sec"
uint8	State //Nagios style 0 ok, 1 warn, 2 crit, 3 unk 4-10 reserved
time_t	Time //the time the event occurred
string	Description //non-numeric state, event, or service description
string[]	Tags //list of tags, e.g., ["sentby:alice","src:nagios"]
float64	Metric //a metric, e.g., the number of reqs/sec.
uint32	TTL //valid time-to-live (in seconds) for this message
string	UID //unique hash or signature from host+service+time+State+Metric

Okay, Let's Kick This Pig

In a perfect world, I could at this point assemble the minions, kidnap the maintainer of every monitoring system, and demand that they import and export this structure for all the relevant events their systems generate. But despite my lack of minions, other problems need solving first, beginning with who pushes and who pulls, and continuing on through wire encoding (protobuf? JSON? XML? etc.), and the litany of details associated with actually putting the messages on the wire, routing them to where they need to go, and figuring out what to do when they get there. So I think, before I can push for native adoption, that there will need to be a fairly well developed model for how data exchange should operate in practice. We need to see what it looks like before we can decide whether it's worth doing.

To that end, libhearsay is a library that implements this common data format and comes with a couple of tools to simplify the protocol and data exchange details. Written in Golang [2] over the past few weeks when I should have been washing the dishes, libhearsay tools employ JSON and Zeromq [3] (sometimes written as 0MQ) to distribute "scraps" of hearsay between monitoring systems (spewers and listeners), enabling your monitoring systems to gossip to each other.

Any monitoring system with something to share can be made into a hearsay spewer with the "spewer" utility. Spewer reads JSON-formatted scraps of hearsay from STDIN or a FIFO, verifies that they are valid (requiring either a host or service name, and a metric or state value), and puts them on the wire via Zeromq push or pub. In push mode, Zeromq will fan out the scraps by fairly distributing them among the connected listeners. In pub(lish) mode, Zeromq will broadcast each scrap to every subscriber.

The generic listener listens to a comma-separated list of spewer socket addresses, and outputs JSON-encoded scraps to STDOUT or appends them to a file of your choosing. The generic listener also has a "Nagios" mode that injects passive check results directly into your Nagios CMD file. Inter-system compatibility will be achieved through the creation of many task-specific listeners that are designed to work with specific tools such as Munin, Reimann, Zabbix, Zenoss, Reconnoiter, Graphite, etc. Each of these listeners will "just work," meaning that given the address of a spewer or several spewers, they'll take scraps off the wire, validate them, and inject them into their parent monitoring system in the way that system expects to receive them.

For now, the generic_listener and some shell scripts can help us get by, and hopefully prove the model, but step 2 certainly centers around the creation of a litany of purpose-specific listeners (some of which should be written by the time you read this). At that point, the cost of entry will be low enough that "normal users" will be able to play. Step 3 will be to push for native support. If you're a project maintainer, expect to see me at your con next year.

Patterns

Zeromq subscribers provide a filter when they subscribe to a pub socket, which enables them to discard the messages they aren't interested in. This should work handily with the "Tag" field in our scrap struct. The model I have in mind for my shop looks pretty much like Figure 2, where all spewers and listeners connect to a central set of redundant message brokers and use filters to extract the scraps from the systems they're interested in.

These brokers are nothing more than a set of systems that have both a listener (to accept scraps from every monitoring server) and a spewer (to copy every scrap back to the interested listeners). Something like a Brooklyn barber shop, all systems know to go to these hosts to both share and receive new hearsay. I imagine that each spewer will use the spewer utilities' "-t" switch to add a tag to each scrap they send, identifying it as, for example: "src:nagios", and each listener will filter for tags of this or that type.

Interestingly, given just the generic spewer and listener tools, any sort of distributed message-passing architecture could be built, and although I'm excited about the possibility of my "smorgasbord of monitoring data" model, I'm even more intrigued to see what other admins might design.

Wait, how does this work exactly?

Let's take a look at the spewer tool in practice by launching it with "-d" to trigger debug mode and sending it a partial scrap like so:

```
[dave@vlasov]--> echo '{"Host":"foo.com","Service":"HTTP",
"State":0}' | spewer -d
Starting Server
got message: {"Host":"foo.com","Service":"HTTP","State":0}
Sending:
{"Host":"foo.com","Service":"HTTP","State":
0,"Time":"2013-07-26T13:51:47.277299512-05:00","Description":""
,"Tags":["Spewed-by:
vlasov.dbg.com"], "Metric":-42,"TTL":60,"UID":""}
```

As you can see, given only a hostname, service name, and state value, spewer created a full scrap by populating default values for Time, Metric, and TTL, and adding a "Spewed-by:" tag, which should help us avoid message loops in the future. If I'd given spewer a "-u" switch, it would have generated an MD5 hash-sum of the message and assigned it to UID.

Spewer also created a 0MQ push socket and placed the scrap on the wire for any connected listeners. If we had a generic listener connected to localhost port 5000, spewer would have read the message and printed it back to STDOUT. If five listeners had been listening, 0MQ would have (round-robin) distributed the message to one of them. If I'd specified "-m pub", spewer would have opened a pub socket and every one of the connected five listeners would have gotten its own copy of the message.

There are myriad ways to get data out of Nagios and into the spewer, but I haven't made a final decision on what interim Nagios support looks like exactly. Because Nagios provides handy macros for things such as hostname, service name, and state, I'm tempted to write a little tool that is intended to be called from a notification command that could inject a scrap into spewer, or modify spewer to accept incoming scraps on a TCP socket locally.

Spewer cannot itself be called via a Nagios command because it needs to persist the publisher socket, and therefore must run as a daemon-like entity. Other options are a Nagios Event Broker module that could inject scraps into spewer, or something as simple as a shell script that could tail a performance log file from Nagios, translating and providing scraps to spewer via STDIN. Each approach has pros and cons.

I also anticipate the need for an indexing service of some sort to enable listeners to find spewers securely. I'll cross that bridge when I come to it.

This may be a long road to a dead-end, but at the moment I'm optimistic and by the next issue expect to have some real systems talking to each other. If you'd like to hack along, feel free to grab libhearsay from GitHub [4] or my blog [5]. Any help would be vastly appreciated and is 100% guaranteed to be repaid in beer at the first convenient conference we both attend.

Take it easy.

References

[1] http://riemann.io/concepts.html.

[2] http://golang.org/.

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

[4] https://github.com/djosephsen/Hearsay.

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