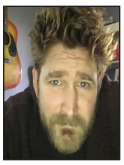
iVoyeur
OpenTracing

DAVE JOSEPHSEN



Dave Josephsen is a book author, code developer, and monitoring expert who works for Sparkpost. His continuing mission: to help engineers worldwide close the feedback loop.
dave-usenix@skeptech.org

As I write this, I am just back from KubeCon and CloudNativeCon [1], where process isolation is a business plan and all your friends work for Microsoft. I freely admit: it was a confusing conference for me in many ways; in fact, trying to get it all down on paper now, I even find the ways in which it was confusing, confusing. Rarely do I find myself so confused that I must engage in the process of attempting to categorize my own confusion, but this is definitely one of those times, so let's see what we can do here.

I suppose the best place to begin is with the ecosystem, which is currently undergoing so rapid an explosion of growth that the Cloud Native Computing Foundation (CNCF) organizers literally had posters on the wall to remind everyone just what the CNCF actually consisted of. Each CNCF project was also given its own space in the keynotes wherein it introduced itself to the user-base, which gave one the sense that a great deal of the current organizational structure had only recently been ironed out. There were also 20 (!) keynotes, covering project updates on 14 CNCF core projects (many of which I was hearing about for the first time). That's ignoring, of course, the parallel explosion of Kubernetes-related start-ups outside the CNCF, all fighting for mind-share, whose founders seem invariably to happen to be former Google employees.

To be clear, I'm making that observation without my tinfoil hat on. To be sure, one might be tempted to infer from the founders homogeneity of pedigree, some greater and possibly diabolical plan, but if such a plan existed I feel like there would be a lot less redundancy among them. Currently there are, for example, 10 competing container-runtimes (at least): Docker, rkt, containerd, CRI-O, LXC, OpenVZ, systemd-nspawn, machinectl, lkvm, and Kata containers. (That's not counting the proprietary container runtimes used by the platform behemoths like AWS, Google Compute Engine, and Azure.)

Speaking of Azure, remember Microsoft? The company that stole all their core products and then spun off BSA [2] to sue everyone for copyright infringement? Remember? They were the ones who anti-competitively buried everyone they couldn't buy, and then sent SCO to assassinate Linux with a copyright lawsuit?

All totally so five minutes ago. At KubeCon, Microsoft is that low-key, tastefully appointed booth toward the back, where a well-spoken, highly tattooed twenty-something is speaking to passersby earnestly and excitedly about the future of open source while handing out rad Golang stickers. As for the other vendors on the floor, I only recognized half a dozen or so. It was like walking the vendor expo in a parallel dimension where Disney is an evil media syndicate hell-bent on owning everything, and Microsoft a benevolent open-source cheerleader and funder of hacky experimental Google code.

And speaking of anti-competitiveness, despite the myriad overlap in functionality between so many of the tools, I never came away with the sense that any of them were serious competitors. I mean, it's pretty obvious you're in competition if you are currently one of 10

possible mutually exclusive container-runtimes for Kubernetes, but having been in the room with them at the runtime salon, I can tell you, the lack of competitive tension between them was, well, confusing.

One thing there could be no confusion about was the CNCF's choice of monitoring tool, emphatically Prometheus [3]. And while, yes, we should talk about that eventually, right now my heart pulls me in another direction: namely, the OpenTracing API [4] project.

Have you read the Dapper [5] paper? Published in 2010, it describes Google's production distributed systems tracing infrastructure. I bring it up because OpenTracing owes its lineage directly to Dapper, so if you really want to sink your teeth in, that paper is probably the best place to start. It's also just a really good read.

I can hear you asking "Really, Dave?! Distributed tracing?!" I know, I know; talk about confusing. First of all, it isn't even monitoring, it's application performance debugging or something like that. And second, it's basically made of magic and impossible for laymen to comprehend, and anyway all the tracing stuff out there is proprietary and expensive. Also, I heard sampling is involved, and anyone who has read anything about monitoring in the last 10 years *obviously* knows that nothing but raw, unsampled, nanosecond-resolution metrics can solve production issues in the real world. MONITOR EVERYTHING HOOYA!

Hear me out for a second, though; I've been doing this for a while, and one thing I've seen quite a bit of is abstraction layers that make monitoring irrelevant. VPNs and tunneling protocols breaking SNMP traps, JVMs breaking systems memory monitoring, VMs breaking process monitoring, containers breaking VM monitoring, and on and on. If the Fundamental Theorem of Software Engineering [6] states all problems can be solved with one additional layer of abstraction, I propose this corollary: *every monitoring system can be made irrelevant with one additional layer of abstraction.*

Here's a heads-up from yours truly, even if Kubernetes isn't the inevitable future of computing everyone says it is; we're in for a *drastic* increase in the use of abstraction layers in the next 10 years. This is an important reason I'm such a big fan of StatsD and the process emitter/reporter pattern [7], wherein we move our monitoring up the stack into the process itself and let the processes we care about emit metrics directly to a monitoring system rather than trying to infer "badness" from system-level metrics. It's difficult for anything to break your monitoring when the programs you care about carry their monitoring around inside them, but even the process-emitter pattern has some abstraction to worry about—namely, microservices infrastructure.

The services design pattern reduces the amount of work that a given process performs. A service is the smallest useful piece of software that can be defined simply and deployed independently (a program that does one thing and does it well), and therefore the metrics it emits are smaller in scope. Instead of, say, one process emitting 10 metrics that expose the entire inner workings of a given job, we now have one metric each from 10 different processes.

Maybe that's fine. If we have a problem that's endemic to one service, it should be easy to pinpoint, but if our problem is the result of a particular call-path or the accumulated latency of many calls to multiple problematic services, we have a correlation problem on our hands. To solve problems with requests moving between multiple processes, we need to know which metric measurements relate to the same individual request.

In many ways, I think distributed tracing acts like a multi-process-aware metrics emitter. Tracing is monitoring; it's just scoped a little differently. Instead of monitoring a host, or a daemon, or an application, we are monitoring requests.

But how do we monitor a request, Dave? Requests are ephemeral. We can't put our hands on them.

Hogwash. Ops has been doing it for decades. Think of the `Received:` header in an SMTP request. Each mail server that has a hand in message delivery knows to unpack and add its own `Received:` line to the email headers. Using those lines, we can dissect the path an email took from sender to recipient, as well as using the date/time stamps to derive latency metrics between hops. Distributed tracing does the same thing to propagate ad hoc metrics between hops by way of the HTTP headers, or whatever other transport is being used.

All we need is a standard that describes the structure of that header, and a collection of language APIs that implement the standard so services can search for, unpack, modify, and repackage the header regardless of the language they were written in or the architecture they run on. SMTP's `Received:` header, along with the rest of the email headers, doesn't work by magic; they've been implemented and reimplemented in every language on every architecture that has ever needed to send an email.

Another interesting aspect of SMTP's `Received:` header is that anything can consume it at any time. The implementation is indifferent with respect to its consumers; rather than being purposefully designed for this or that sort of introspection system, it can use anything that knows how to unpack and parse it.

Like SMTP's `Received:` header, the OpenTracing project provides a consumer-agnostic means of tracing individual requests through large, high-volume distributed systems. It's implemented as a header that piggy-backs along as a request makes

its way through a distributed application. OpenTracing provides APIs in nine languages, which makes it trivial for you to unpack, modify, and repackage the header without having to worry about the wire-protocol details.

Unlike SMTP, distributed application requests aren't linear by nature. Your request to `/foo/events` might spawn subsequent requests to `/foo/user-events` and `/foo/app-events`, for example, along with one or more database requests to look up user-IDs or authorize your request. When one request begets another that it depends on, OpenTracing describes that relationship in terms of parents and children. When a request begets another that it doesn't depend on (say a non-blocking callout to a logging service or a cache-write), OpenTracing describes the relationship as a "FollowsFrom" relationship. Each individual request (or operation) is described by a `span` struct, while the relationships between individual spans are maintained by a `SpanContext`.

Your job as a user of the API is to instrument your code to create a span roughly at each process boundary (wherever a request is sent or received and at exit). Within each span, you can create tags to track metrics like wait times or log process details.

My mention of database calls in the paragraphs above was intentional. How can we hope to meaningfully trace requests that cross process boundaries into binary monoliths like MySQL or Cassandra? To be sure, we can time our DB interactions from the client-side, but everything happening inside the DB is a black box to us.

The good news is, given that OpenTracing is an API, support for it is slowly being proliferated into web-frameworks like Flask, RPC-layers like gRPC, databases like Cassandra, and even web-servers like Nginx. These tools all fully support existing OpenTracing SpanContexts today, automatically unpacking them and adding new spans to provide a uniform source of insight into critical processes that have historically required vastly different monitoring strategies.

Confusingly (but on-message with respect to the greater Kubernetes community), there are nine (!) different tracer implementations that can be used to inspect OpenTracing data: Zipkin, Jaeger, Appdash, Lightstep, Hawkular, Instana, sky-walking, inspectIT, and stagemonitor. Some of these are language specific and others proprietary. Jaeger, Zipkin, and Lightstep are all good places to start for generalists.

I'm kind of in love with the OpenTracing project's goal and implementation, and I hope I've done both of those justice in this introduction. Tracing is monitoring, and it's not made of magic, though it is somewhat magical. I'm really looking forward to API support in tools like Ruby-Rails and Node, and if I can get things arranged to be able to afford the time, they're on the top of my list for OSS contributions in the new year.

Take it easy!

References

[1] KubeCon + CloudNativeCon: https://events.linuxfoundation.org/events/kubecon-cloudnativecon-north-america-2018/.

[2] The Business Software Alliance: http://www.bsa.org/.

[3] Prometheus monitoring software: https://prometheus.io/.

[4] OpenTracing API: http://opentracing.io.

[5] Dapper paper: https://research.google.com/pubs/pub36356.html.

[6] FTSE: https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering.

[7] Process emitter/reporter pattern: http://blog.librato.com/posts/collector-patterns.