Editor: **Robert Blumen**
SalesForce
robert@robertblumen.com

Ben Sigelman on Distributed Tracing

Robert Blumen

From the Editor

We bring you this month one of my own shows, Software Engineering Radio Episode 337, featuring guest Ben Sigelman. Sigelman is the cofounder and chief executive officer of LightStep, where he is building reliability management software, and a coauthor of the OpenTracing project. We discuss tracing in general and distributed tracing, which involves the propagation of tracing across process boundaries in a distributed system. The discussion covers the basics of tracing, how distributed tracing is different, the instrumentation required to collect tracing data, what is collected and how, where the data go, and use cases for the data itself, including monitoring, analytics, and capacity planning. The excerpt presented here contains about one half of the interview, with the remaining half available for download from our website or via RSS.—*Robert Blumen*

Robert Blumen: Today we're going to be talking about distributed tracing. First, I'd like to talk about tracing in single instance and then we'll move on to look at the complexities of a distributed system. To start, what is tracing?

Ben Sigelman: The industry has used the word *tracing* to refer to things having some commonality but are quite different, like stack traces (which have that word *trace* in them), kernel traces like DTrace, and distributed tracing systems like Dapper and Zipkin.

Digital Object Identifier 10.1109/MS.2018.2880598
Date of publication: 8 January 2019

In all of monitoring and observability, there are fundamentally two types of data: event data and statistical data. Tracing data are definitely event data.

Kernel tracing events take a couple of nanoseconds and happen very frequently. In DTrace, e.g., you write a script (that looks like an awk script) that takes as input a high-frequency event stream and generates useful statistics. Stack traces trace up the stack. There is nothing mysterious about that, but it's a totally different thing than kernel tracing or distributed tracing.

In distributed tracing, you're looking at a single logical transaction in a [distributed] system.

Let me clarify. Statistical data could be things like amount of memory used, CPU load; whereas event data consist of particulars about what happened in each case. Is that the difference?

Event data has a time stamp indicating when the event occurred. CPU level is an aggregate. It is a statistical summary of what happened during a time period, e.g., 75% of the time your CPU is in a runnable state. Other common statistics would be things like event rates and latency percentiles.

We had another episode of Software Engineering Radio dealing with logging. This sounds like it might

be similar but different. What is the distinction?

Logging is very much on the side of event monitoring. That word is also really problematic in that it means different things to different people, but I think logging has come to mean: event monitoring where the cost of centralization is reasonable. You pay to centralize your logs and then search over them. It would be great if you could get every single system call into your single combined [log aggregator], but you can't because it's too expensive. So you don't consider that part of your logging strategy.

Could you drill down into more detail about the thing we call *tracing*? What differentiates that from other types of events?

If we went back 15, 20 years tracing meant *automatic* instrumentation of things that usually happen at the kernel boundary. Every system call could be traced, or every function call, whereas logging used to imply logging statements added by programmers to their code.

Then tracing is a cross-cutting concern, where something can intercept the normal operation of your system and collect event data from that?

Yes, and this gets very blurry when you start talking about more modern architectures, but if we're talking about the historical context, I think that's totally accurate.

In a modern system, what are some of the primary use cases for tracing?

Modern means that you're building software that involves many teams who are working in concert, developing services, serverless, or micro services. In that kind of system, tracing is moving up the stack considerably because the pain point people are having is also moving up the stack considerably.

If you're dealing with a distributed architecture, the questions you're trying to answer are very primitive—things like: "This transaction was slow," or, "This transaction had an error. I have no idea what happened. I have no idea what services it touched. I have no idea what happened in those services." You are trying to figure out which services were involved. You would be thankful if you could *get* to the point where you are trying to understand what happened at the system call level. And that's a very different question.

Distributed tracing lets you see across system boundaries. If service A calls B, calls C, calls D, and D is having a bad day, it affects service A. [With distributed tracing], you can figure that out very easily. And that's a profound thing if you didn't have it before.

If you're either trying to figure out what happened in one case, or why something taking as long as it did, then tracing is primarily a troubleshooting or debugging tool. Is that accurate?

I think the answer to that is changing. Historically, distributed tracing has been used for performance analysis and for root cause analysis. Making performance improvements, you want to focus your energies where it's going to have an effect on the business.

When I was at Google, before we deployed [a system], we had people spending six months on 20% performance improvements that were off the critical path for the end user. User latency was not affected by these performance improvements because they were the wrong place. It made no difference. Distributed tracing can help you avoid that failure mode.

Making performance improvements, you want to focus your energies where it's going to have an effect on the business.

The other use case is you're woken up at three in the morning. You know that something bad is happening, and you need to figure out where as quickly as possible. If you are working in a distributed environment, a tracing system can be really helpful with that. In both cases you're actually looking at individual traces to make these assessments.

Distributed tracing data have a fount of knowledge about how these distributed systems interact. From that we can obtain higher-level insights for developers, operators, and management to better understand these systems. It's going to be a lot more powerful than looking at individual traces in a UI.

You mentioned critical path analysis. Explain what that is.

In a modern environment, there is a lot of concurrency. You do things in

parallel because if you called everything in serial, it would take days to get back to users when you have hundreds of services.

If you call out to five services in parallel, and you wait for all of them to return, the one that comes back last delays the end user. The one that comes back last is on the critical path. You need to understand what the laggard was and focus your analysis there. It's an easy thing to if you have the structural information, which distributed traces do.

A good tracing system should help you surface the critical path automatically so you don't waste time analyzing things that aren't holding up the end user with a transaction at the top of the stack.

Operations may not be aware of all of the changes in a system that were deployed in the last week. Is tracing a good reverse engineering tool for operations? Like for finding out what services call A or are called by B?

I would argue it's the best reverse engineering tool. That's why people get excited about it. The value prop is when you need the information it's right there in front of you. That's almost the definition of what it's doing.

We did another show about latency and latency outliers. Latency outliers are far more important than average latency for human perceived responsiveness, but these are events by definition don't occur very often. Can distributed tracing identify the 0.1% worst performing requests?

Absolutely, especially if the tracing system is designed to focus its energies in those areas. In my experience, if you have a system where latency degrades without a software release, almost all of the time it is because something is overloaded. Most sudden production latency regressions are due to throughput.

You have some kind of overwhelming throughput in a system, and that creates a bottleneck. In queuing theory, if there is a bottleneck that is reflected in high latency. Now you need to understand where that load came from. For example, if you have a storage system and the CPU is getting really hot, it's probably because of consolidation and batching. Understanding that requires looking at all of the other requests that it is serving.

Requests can be thought of in isolation, but the requests from a couple of minutes ago that are getting batched and are causing you to saturate CPU are actually affecting the latency for requests that are happening minutes later. Understanding that kind of root cause analysis is very, very challenging, because the amount of data are so overwhelming that if you centralize all of it, you're not going to be able to afford your observability system, and if you don't you literally lack the information to run that analysis.

Now we're going to drill down into more detail about what goes on in collection. I want to go through some definitions that will enable us to have this discussion. There are concepts that are in the literature that I want you to explain what these are, the first one being a *transaction*.

I think a transaction ought to be considered a single logical unit of work, in its entirety. I say "in its entirety" to emphasize the fact that the same transaction may move from process to process, from machine to machine, and from thread to thread.

Can give an example of a transaction, either something you worked on at Google or at another project?

Sure. Web search is an example that we can all relate to. You type your query. The transaction begins in your browser. It goes through a number of front-end layers. Google runs a web server, which is where the logic starts. That farms it out to dozens of services that all take a crack at your query. Google takes the results and combines the results into a result set that it presents it to you.

I want to move on to another important concept in tracing, which is the idea of *context* and *propagation*.

There is a sequence of events that affect a single transaction. The context object is usually the thing where you store some kind of central identifier that you can use to tie that sequence together. We do things concurrently so the context literally splits in half and then rejoins later.

Let's go back to this Google search example. What are some interesting fields that would be in the context?

The approach that we took with Dapper was honestly pretty simplistic but was effective, which was to have two unique IDs. One was called the *trace ID*, which lived for the entirety of this one transaction, and the other was called a *span ID*.

A span is one logical segment of that trace that doesn't have any internal forking and joining of its own and is the right size to measure in a system like this. You're not going to have a span for every system call, but you probably will have a span for every RPC or HTTP call.

The trace ID is consistent for the entire trace. The span IDs are unique within that trace. You form a tree, more formally theoretically a graph of spans pointing to their parents. That allows you to infer the structure of the transaction and identify things like the critical path. The context will contain the trace and span ID.

I'm hoping to get something more concrete like the name of each service it visits, the IP address, maybe this stack trace or call stack on that process.

That's not in the context. There are two types of data that you want to record. One is the data that you record in band. If you're sending a request from service A to service B, you have to pass along some context in band with application data. The in-band data are very small. All you want to do is record unique IDs. In Dapper we recorded a trace ID that was consistent for the entire transaction and what we called a *span ID* that represents that one service call.

The other data are out of band. In the out-of-band channel, you record much more detailed information: timing, tags, names of services, names of endpoints, even a micro log of events that took place for each span. That's all sent out of band, buffered, and does not need to happen in real time. You get it out of the process as efficiently as you can.

There is this thick buffered out-of-band channel and a thin in-band context, which just record unique IDs.

This sounds something like how log aggregators work, where you do not need to forward the log message to the log aggregator during the work the program is doing, as long as it gets queued up and later gets sent.

Exactly the same.

I'm inferring these IDs enable you to correlate all of the different collections that occurred across many different servers, so that you're able to match up all of these different pieces of context.

Exactly.

Do we trace exceptional conditions or errors as well?

We should. It's a goal for a systems like this to have extra detail when things aren't going well. An error would be an example of that. Both soft errors and hard errors.

ABOUT THE AUTHOR

ROBERT BLUMEN is the lead DevOps engineer at SalesForce Desk.com. Contact him at robert@robertblumen.com.