**EPISODE 886**

[INTRODUCTION]

**[00:00:00] JM**: During 2015, Uber was going through rapid scalability. The internal engineering systems of the ridesharing company were constantly tested by the growing user base. Over the next two years, the number of internal services at Uber would grow from 500 to 2,000, and standardizing the monitoring of all these different services became a priority. After working with a variety of available tools, Uber's engineering team decided that something new needed to be built internally. Jaeger is an open source distributed tracing tool that provides observability features throughout Uber's microservices architecture.

Yuri Shkuro is an engineer at Uber where he works on Jaeger and other infrastructure projects. Yuri joins the show to discuss the history of engineering at Uber, the architecture of Jaeger and the requirements for building and scaling a distributed tracing tool. To find all of our episodes about Uber and other scalable engineering organizations, you can check out the Software Daily app for iOS or Android and you can search for Uber and find those episodes.

[SPONSOR MESSAGE]

**[00:01:15] JM**: Monday.com is a team management platform that brings all of your work, external tools and communications into one place making cross-team collaboration easy. You can try Monday.com and get a 14-day trial by going to Monday.com/sedaily. If you decide to become a customer, you will get 10% off by coupon code SEDAILY.

What I love most about Monday.com is how fast it is. Many project management tools are hard to use because they take so long to respond, and when you're engaging with project management and communication software, you need it to be fast. You need it to be responsive and you need the UI to be intuitive.

Monday.com has a modern interface that's beautiful to look at. There are lots of ways to use Monday, but it doesn't feel overly opinionated. It's flexible. It can adapt to whatever application you need, dashboards, communication, Kanban boards, issue tracking.

If you're ready to change the way that you work online, give Monday.com a try by going to Monday.com/sedaily an get a free 14-day trial, and you will also get 10% off if you use the discount code SEDAILY.

Monday.com received a Webby award for productivity app of the year, and that's because many teams have used Monday.com to become productive. Companies like WeWork, and Philips and Wix.com. Try out Monday.com today by going to Monday.com/sedaily.

Thank you to Monday.com for being a sponsor of Software Engineering Daily

[INTERVIEW]

**[00:03:08] JM**: Yuri Shkuro, welcome to Software Engineering Daily.

**[00:03:11] YS**: Thank you. Nice to be here.

**[00:03:13] JM**: Yeah, it's nice to have you. You've been at Uber since 2015. You've seen the rapid scalability of the company. Describe your experience back when Uber was going through that early hypergrowth.

**[00:03:26] YS**: When I started, I kind of started working on tracing almost from the beginning. At the time, we had roughly a few hundred microservices, I think 500 was the upper bound, and roughly the same number of engineers ironically. Yeah, within a couple of years, it exploded to several thousands. Today we don't really count them, but it's about 3,000.

So, yeah, that was kind of an interesting transition where a lot of functionality moved out of the old to monolith into microservices and the resulting complexity that was created is actually a question how much microservices are beneficial versus the resulting complexity in my mind.

**[00:04:08] JM**: Really? Okay. Can we go a little bit deeper on that? Let's start off with controversy.

**[00:04:13] YS**: Yeah. Well, microservices are, in general, kind of hard thing to do right as many people know, and there are many aspects to that. The basic ones that people typically think of just the infrastructure concerns, like how to do service discovery, deployments, sort of management, multi-zone, multi-region setups and all of that distributed system stuff. That's obviously takes a lot of infrastructure resources and teams to build, and there aren't really great examples in open source still. It's kind of you can probably piece together from multiple different products.

But when we were starting, almost all of Uber infra is internally built. Not counting like data stores, and maybe Mesos is one of the like biggest infra parts, which is not an in-house build, but a lot of stuff is built manually because there was no real good scalable alternatives to microservices. So that's one aspect. But the other aspect that people don't often talk about I think is the actual complexity that we create, and with complexity come a lot of other issues. One of them was reliability, because now that – Everyone knows if you ever read the new distributed systems paper, it says, "Well, communications are not reliable. It's a given."

So microservices' lens just double bet on this manual communications to build very complex systems. The reliability of their old system is typically much worse, because you increased the number of failure modes exponentially. So something needs to be done about this. The actual design of the system needs to account for all of these things. So that's extra complexity.

I guess like my close to home thing is the observability of this whole system is harder without tools like distributed tracing. So I can go more on that. It's really I think is a big topic that people don't realize that microservices are always painted with like rosy brush to me.

**[00:06:03] JM**: Yeah, let's talk about that a little bit more. We will get into distributed tracing and Jaeger and so on. But I've heard a couple of other very experienced engineers tell me some form of this that maybe we're going down a path of insanity. Maybe we shouldn't be doing this whole microservices thing. Maybe we should all be figuring out how to run monoliths better. Is there an alternative to microservices?

**[00:06:30] YS**: There is definitely an alternative. I think the monolith is obviously an alternative which works, because Facebook is still run in monolith for a large part of their web presence.

We know that it's doable. It scales. It has the rapid deployment. So one of the big reasons people go to microservices adaption is that it allows you to scale your organization better, because you focus teams working on like smaller parts of the system so that they are not affected by the other parts, which sounds good. I mean, that's probably true, but there is a flipside to that, is that first of all, none of the components in the system are really working by themselves.

Yes, you introduce a certain kind of autonomous behavior to a team, but at the same time they're not solved in the overall business problem by themselves. Especially if they're somewhere in the middle of the stack, like a payment system or whatever, fulfillment system. They still integrate with a lot of other stuff. And integrations are actually harder in that way than if it was all in a single codebase and in a single application potentially.

I guess I don't know if there are other alternatives. I mean, monolith doesn't have to be like one single monolith for the whole company. It could be potentially like several big monoliths. Maybe partitioned by business domains. But there's still a huge difference between having – I don't know, five monoliths in the company versus 3,000 microservices.

**[00:07:56] JM**: So in this period of time when Uber was breaking up its monolith and it was moving to microservices. I believe that from 2015 to 2017, the number of services at Uber grew from 500 to 2,000. Describe how Uber's infrastructure evolved over that period of time.

**[00:08:15] YS**: So when I just joined, when we were still doing effectively pre-allocating hosts to individual services and they were running on bare metal, which is a very kind of manual way of doing deployments. Because if you hose those down sort of to your capacity of the service or whatever application is decreased and you kind of have to go manually, restart it or do something about it. So there's a lot of operation overhead with that approach obviously, and we haven't had at the time things like Mesos or Kubernetes. So that was one big push to that, which I think in a way helped with developing microservices, because in that old world, it was actually much easier to just add functionality to an existing service, because you didn't have to deal with all these provisioning and infrastructure setup for a new cluster of hosts.

 

So, Mesos, once we migrated to Mesos, that kind of went away because it gives us dynamic allocations, which is arguably a much better situation in any case no matter how you deploy the applications. The other part that we were working very heavily on was equivalent of a service mesh. I think, traditionally, back even before I joined Uber was routing most of the traffic with HAProxy, which was find when you had the static pools of hosts. But as we were moving to more dynamic placements of services that, like with HAProxy, it wasn't that easy to actually update configs on-the-fly and there was not good mechanism for that.

I mean, we built something, but eventually we deprecated that in favor of a real service mesh. With a service mesh, Uber had an interesting story. We developed this internal product called Hyperbahn, which didn't do well. One of the reasons is because it was implemented in nodes and it obviously had performance issues because of that. Also, by design, it was kind of a tradition more hops than the normal service mesh would do.

So, we eventually migrated to more or less what people think of service mesh today, except that we're not running on service mesh agents as a sidecar. We're running them as a host agent. That's why it's not a real true service mesh, because the host agent, it's efficient, because you have only single hop. Your service talks to the local host agent and then that agent forwards directly to the destination service, so one extra hop on a network.

But because it's an agent, it doesn't really know things like, "Well, who is calling me? What's the identity of the caller?" So it is not able to support things like service authentication explicitly. So you could do it on the application side, but not through the infrastructure. So even today we're still – That's a work in progress. We're migrating to more like a true service mesh similar to Envoy, which runs as a sidecar and then allows us to do more things on the infrastructure component rather than pushing them into your application space.

**[00:11:09] JM**: Yeah, we did a previous show about that. So this idea of the service proxy or the service mesh, this is a standardized library or sidecar that rides alongside your service. The service proxy or the service library, it fulfills some standardized sets of roles that you would want along with every service. It might give you observability features, or rate limiting features. How have that suite features that you've gotten out of the service proxy, how have those evolved over that period of time that you've been at Uber?

**[00:11:53] YS**: So, I think primarily we were focusing on sort of the routing function of the sidecar. Meaning, that it takes care of service discovery. We also build a fairly advanced, some people say more advanced than let's say Istio, control plane, which allows to do things like automatic rerouting of traffic to different zones, to different regions even. You can drain traffic gradually from one location to another. So those are the main routing functions that our service mesh performs.

I'm actually not sure if it does the rate limiting. The previous version, the Hyperbahn did try to do rate limiting, and it was a challenging problem as well, because it sometimes didn't work well. So, in terms of observability, mostly what we get from service mesh, the metrics, like service-to-service metrics for RPC calls, which is very handy, because we want to build internally a tool in our observability team, a tool which automatically detects those metrics. If you're a service owner, you come to a page and suddenly it populates a whole dashboard for you with whole tons of infrastructure metrics submitted by various infrastructure components that we know your service is using. It's like auto detected, which is very awesome, because you don't have to generate this dashboard or manually configure them. They're very standardized and auto discovered. So, the service mesh metrics is one of them. It just comes up and you can get a whole bunch of like error counts, how many calls to which targets you're doing and things like that.

**[00:13:30] JM**: So whoever stands up a service at Uber, whether they're standing up a microservice under Uber Eats, or they're deploying a new pricing algorithm that is in its own service. Everybody wants a certain stack of features. Everybody wants distributed tracing, for example. Let's talk about distributed tracing, because that's what you're an expert in. Explain what distributed tracing is.

**[00:14:03] YS**: Distributed tracing is observability technique that is different from many other techniques, and that it monitors not behavior of a single component of the architecture, but rather behavior of the single request, which is executed by the overall architecture. So think about like a request to Netflix homepage. It actually hits, if I remember, about 20, 25 different microservices, right? At Uber, a similar situation, the request for driver go online from the mobile

app as it comes to the data center, it hits about 30 microservices doing around 100 RPC calls as part of that one single request execution.

So what distributed tracing does, it actually looks at – It traces this whole request through all these components of the architecture and gives you various information about it, like timing, causality, who called whom. You can make it as rich as you want, really, because the instrumentation supports like tags and logs. You can attach timed events to the data. But at the bare minimum, you get sort of the view of the whole request for the system.

It's almost like a distributed stack trace. If you compare it with a monolith, if you have an exception somewhere in a monolith, you get back a stack trace, which gives you the exact path that a request took to that point. That's kind of what the distributed trace has accepted. So it's not just one single sequence. It's actually a tree of different calls that you make on multiple services.

[SPONSOR MESSAGE]

**[00:15:39] JM**: When a rider calls a car using a ridesharing service, there are hundreds of backend services involved in fulfilling that request. Distributed tracing allows the developers at the ridesharing company to see how requests travel through all the stages of the network. From the frontend layer, to the application middleware, to the backend core data services, distributed tracing can be used to understand how long a complex request is taking at each of these stages so the developers can debug their complex application and improve performance issues.

LightStep is a company built around distributed tracing and modern observability. LightStep answers questions and diagnosis anomalies in mobile applications, monoliths and microservices. At lightstep.com/sedaily, you can get started with LightStep tracing and get a free t-shirt. This comfortable, well-fitting t-shirt says, "Distributed tracing is fun," which is a quote that you may find yourself saying once you are improving the latency of your multi-service requests.

LightStep allows you to analyze every transaction that your users engage in. You can measure performance where it matters and you can find the root cause of your problems. LightStep was

founded by Ben Sigleman, who is a previous guest on Software Engineering Daily. In that show he talked about his early development of distributed tracing at Google. I recommend going back and giving that episode a listen if you haven't heard it. If you want to try distributed tracing for free, you can use LightStep and get a free t-shirt. Go to lightstep.com/sedaily.

Companies such as Lyft, Twilio and GitHub all use LightStep to observe their systems and improve their product quality.

Thanks to LightStep for being a sponsor of Software Engineering Daily, and you can support the show by going to lightstep.com/sedaily.

[INTERVIEW CONTINUED]

**[00:17:50] JM**: Because you have this standardization across different services at Uber, thanks to your service library or your service proxy, I'm guessing that at this point it's safe to assume that every service is going to get instrumented properly with the distributed tracing infrastructure out of the box.

**[00:18:09] YS**: Actually, no. This is a big misconception about service meshes I think. If you read the fine print of every service mesh, they say that, "Yes, you can get distributed tracing from them provided that the application forwards a certain headers with some context."

In my experience, the forwarding actually is the hardest part, because it does require instrumentation in the application. You cannot just take an application as a black box and connect them by the sidecar and try to get a trace out of it, because the reason is very simple. If you have 10 requests coming into your application and each of those requests makes 5 downstream calls. So you have 50 requests going out of your application. How do you correlate those 10 on the inbound and 50 on the outbound?

The service mesh cannot do that as basically treating an application as a black box. You need something in the application that actually provides that correlation, and that's what instrumentation for distributed tracing is primarily doing. Projects like open tracing and open telemetry that I'm deeply involved with. They give you like a standard instrumentation that in

many cases you don't have to actually write any code. You can just attach a JAR to your process or do some bare minimal instantiation of a tracer. But once you've done that, then yes, you get full tracing.

At that point, ironically, the tracing that – Additional tracing that can be provided by the sidecar becomes sort of not that useful. It gives you an extra hop in a call graph, but you don't need that. We're actually not using that at Uber at all. The service mesh does not provide tracing.

**[00:19:42] JM**: Let's go back to the point at which you joined Uber in 2015. Did you originally join to work on the distributed tracing team?

**[00:19:52] YS**: No. I joined to work on the infrastructure overall, because before that, I was working in investment bank mostly on the product side. So I kind of wanted to try out infrastructure work. I was interested in that. In New York, there was only like 10 people in New York office at the time and the only infrastructure in New York was observability, and specifically the metrics team. So they were working on what later became known as M3DB, our open source metrics database.

So I joined that team, but they were kind of done at that point with the first version. So they didn't necessarily need my help. So I started looking for other projects in the observability space, and distributed tracing was one big gap that no one was actually doing at Uber. So we decided to make it the mission of our team.

**[00:20:41] JM**: Okay. When you started that distributed tracing team focus, what was the lay of the land in terms of tooling at Uber and open source tooling that was available. What was the set of things that you were able to pull from to start to think about what Uber's distributed tracing stack would look like?

**[00:21:04] YS**: There are a couple of things which affected the direction. One was that Uber in the previous year maybe before I joined built its own RPC framework and protocol called T-Channel. I mean, you can think of it somewhat similar to GRPC. It's its own binary protocol with its own frame format, etc., and it was implemented in multiple languages using Thrift as an encoding.

One of the features of that overall design was that they've envisioned tracing been built in into the client libraries of that RPC framework. So there are not a lot of services at the time using that, because there were some. So we already had sort of distributed tracing instrumentation in a bunch of services at Uber. But what we didn't have is a tracing backend that would collect them. But there was like a prototype backend running using the Zipkin server with some other custom components built in Node.js and like using React as a store, I think. So it was kind of very non-standard set up of Zipkin.

So what we've done is we decided, "Well, first let's build a proper kind of production worthy tracing backend." That's what we've done. We didn't have user interface experts in our team at the time. So we decided, "Well, we'll just use Zipkin frontend for this." But because the T-Channel protocol was already sending data in a custom format, which wasn't Zipkin. We had to build some collection pipeline that would store data in a way that Zipkin could understand. So that was effectively the birth of Jaeger, is the collectors that were receiving all these spans from T-Channel and storing them in Cassandra in a way that Zipkin could then read them to display. So that's how it started.

Then once that was in place, we were kind of covered for a while on the T-Channel side, but a lot of services at Uber was still communicating using plain HTTP. Sometimes with JSON format, sometimes with Thrift, but not using any specific framework other than whatever the language provided, like sometimes very low-level HTTP frameworks in the language.

So, we needed an instrumentation for those services as well, because they were not traced at all. That's where we realized that there is no real good solution in open source which would be some sort of a standard. So we could obviously try to use Zipkin client libraries for that, but Zipkin libraries were in an interesting state, because they were not officially maintained by the Zipkin project. Maybe Java, like a brave library was partially at the time maintained by Zipkin project. Eventually it became full sub-project.

But all other languages were done by people ad hoc. Some person would write a Python version of like a Node.js version, and there was a lot of inconsistency between those libraries in

terms of what terminology they used. How you interacted them. What API they used? The quality of those libraries was also very varied.

So we wanted – If I was to go to all the service developers at Uber and say, "You have to instrument your services with something," I would have preferred to have some sort of a standard and official open source API that I could give them to do that. That's how open tracing was effectively born, because when I went to a Zipkin workshop, there are like a dozen people there and they all had that exact same problem and they said, "Well, why don't we do just that?" Then we started open tracing, which would provide you very unified way of putting instrumentation into your service without worrying about what tracing backend you're going to use. What even tracing library you're going to use. As long your application code speaks to a certain API, you're covered.

So that was kind of the next evolution of our tracing team and our work, because from that point, we spent some time building client libraries for Jaeger that would support open tracing API. The way the API was developed, it evolved into a structure, like a data structure, which was slightly more rich than Zipkin was supporting. So, we had to invent our own data model in Jaeger, our own storage. So eventually like Jaeger became essentially completely separate from Zipkin, because just like all the components were different at that point.

**[00:25:31] JM**: So you touched on some vocabulary there that I'd like to clarify and I'd like to just give a quick overview for distributed tracing. We've done some shows previously about distributed tracing. We've probably done two or three shows about it and people who are unfamiliar should definitely go back, because this will probably be an advanced distributed tracing show.

But I just want to explain how I see distributed tracing. You can tell me where I'm wrong. In a typical trace, you might have three or four services and a request to service A might need to make a request to service B. Then service B might need to make a request to service C. Service C might need to request service D.

If you are just the person who is making the request to service A and there's some kind of failure along the way or some kind of latency, you don't have any idea what caused that failure without

distributed tracing. So distributed tracing can give you the series of spans. So every time this request is propagated through service A and then B and then C and then D, the requests will include some kind of trace ID so that this trace ID is propagated through the different services. In that way, we can figure out how long is the service request that's chaining through this different services, how long is it spending in each of these downstream services.

Then the end result of that trace is these different spans. Each span is some period of time that was spent in one of the services in the chain. So if you have service A, B, C and D, you might have four spans associated with that. Is that a good basic overview of distributed tracing?

**[00:27:22] YS**: Yes. Conceptually, it's very close. I wouldn't necessarily say that distributed tracing has to use a notion of spans. It's better to think of it as there are certain number of events happening in each service. Some of them purely internal. Some of them events like I send a request to another service, or I receive the response from another service. So those would be two events.

So span is just a simplification of that event model where you group the start and end events of certain operation and you call them a span. But, conceptually, it's still underline. There're like certain events, certain trace points in your application where you grab the trace ID that we mentioned and you grab sort of another piece of information, which is a causality. Who was the previous event that led the execution to your point?

So, you construct a graph, direct at the cyclic graph of those events. Again, if you're thinking about a span model, which is a simplification, then, typically, for RPC graph it would be a tree, a very simple tree. Whereas like with – There are other tracing systems which are actually using like raw events. In those events, it may not necessarily be a tree. It could be actually a graph with like – It's not going to have loops, but it's going to have – Sort of it's not a tree.

Yeah. I mean, we're all I think pretty cool. There's also things like if you think about a typical server which receives a request and sends downstream requests, then even in a span model in a situation where you just described A, B, C, D, you might have more than one span per service, because you typically have a span for your inbound request where you received it then you send a response. Then you would have a so-called client span for every call you make

downstream. Again, there's a start when you receive the respond back. So, typically, this is a big more complicated. But on a high-level, you're pretty close.

**[00:29:15] JM**: Got it. Okay. So you were giving us a little bit of history about how the open tracing project got started. Could you refine what you said about open tracing? Why did open tracing get started and how does open tracing relate to the Zipkin project?

**[00:29:32] YS**: Sure. So, Zipkin project is primarily a tracing backend. So is Jaeger. So is Amazon X-Ray or Google Stack Driver or many of the APM vendor solutions. So, typically, the backend, they don't really care how you get the data to them as long as you get the data. Then the backends are responsible for displaying it, to visualizing it, aggregating some, like doing maybe even machine learning, alerts and things like that. But they don't really care about how that data is extracted from the application. That's the part of the instrumentation in an application.

Instrumentation, it's a kind of a bridge between your application and so-called trace points, where these events that I just described get captured. So your application tells the library they're doing tracings in, so I just send a request, "Oh, I received the response." So that's the interaction.

So, the open tracing is the API for that interaction. It says like if your application wants to talk to or sort of send event to a tracing backend. Well, you could use a very bespoke library. You could pick a Zipkin Java library and talk to its precise API that that library defined. Then the tracing data that you collect from your request will be flowing into Zipkin backend.

But if you do that, then you're kind of tying yourself to Zipkin backend or at least to Zipkin data format. So, if you want to switch vendor or like in other open source tracing system, then it's pretty expensive to change your instrumentation. Because now you have to go and, in all those trace points, like call a slightly different API potentially. Even though it may be the same data that you're passing, but the API may be called instead of start span, it's called begin span. So there's like silly changes that you have to make, but there are lots of them. In some cases there might be conceptual changes as well. But most tracing systems are actually pretty close in that regard in the conceptual data model.

So the open tracing was designed to abstract those differences away so that you have a single conceptual data model and a single API that your application talks to, and that beyond that API, whatever implementation you plug in would correspond to the tracing backend that you're using. So, if you want to use Zipkin backend, you would plug in a tracing library for Zipkin which implements an open tracing API, or you do the same thing for Jaeger or a stack driver.

So that's the open tracing API. It's almost like the equivalent of it would be for people who are in Java space, SLA for J. It's a very standard API in Java that almost everyone uses. Because all it does, it says, "This is how you log information." It doesn't tell you what happens to that information after you logged it. You actually need to instantiate the specific logger, which could be like SLA for J. It comes with a simple implementation, but there is also a log back, a log for J. There is a very rich login library. So you could instantiate those. But your application doesn't care at that point and it's like you already written all the log statements that you needed to write. They're not going to change just because you flipped a log in instrumentation. Sorry, a log in instance. That's the same approach that open tracing do. Open tracing is an API that your application interact with. Then whichever like implementation you instantiate is specific to what tracing backend you're using.

**[00:32:54] JM**: Okay. At this point, the listener probably understands that there're a number of different components that fit into distributed tracing. Every application, every service needs to do some work to be sending its information to the distributed tracing library that is sitting adjacent to that service. Then that tracing information has to be aggregated somewhere. It has to be stored somewhere.

In order to do all these work, there's an architecture that fits together as Jaeger at Uber. So the Jaeger architecture includes the Jaeger client, the Jaeger agent, the Jaeger collector. It includes Cassandra, I believe, which is storing some of these traces. Give an overview of the Jaeger architecture.

**[00:33:42] YS**: So, Jaeger project, because we were involved in open tracing from the beginning, we decided that we're going to draw a line between what we provide as part of the

project, and having open tracing sort of project in place was very useful, because we didn't have to spend cycles on implementing the actual instrumentation.

Let's say your service is using GRPC framework. So you need to have some sort of middleware build for GRPC that captures the events from GRPC and pushes them into an open tracing API. So, we didn't have to write it, because open tracing API is open source, GRPC is open source. So the middleware can be written as open source by someone.

The only thing we needed to write was sort of the implementation of the open tracing API, and that's what you refer to as Jaeger clients. So those we did have to implement. That actually is now changing with open tracing – Sorry, with open telemetry project, because that one will come with a standard implementation. So in the future, we might de-scope Jaeger slightly eve more and say, "We don't need to keep maintaining the Jaeger clients, because there's not that much difference in them compared to open telemetry standard SDK."

But what the Jaeger clients are doing, they take in this open tracing the API calls. They converted it into Jaeger span data and they ship it to Jaeger backend. Now in terms of Jaeger architecture, there are multiple ways actually that you can deploy Jaeger. The way we deployed at Uber is we have an agent running on every host. The benefit of the agent is that your client libraries don't need to be configured to know where the Jaeger backend actually lives. They can just send to a specific port on a local host, and that's it.

So the configuration is actually there's zero configuration for Jaeger client libraries at Uber in production. Then the agents, when they deploy it, they kind of have their own infrastructure and they know where to find the collectors.

But the alternative way is that you can also reconfigure your client, Jaeger client, to send data directly to the collector using HTTP request, for example, and Thrift encoding. So that's an alternative, but then you kind of need to know whether the collectors are – Where are the locations that you probably need some sort of either virtual IP or a DNS name and do some load balancing on the backend. So we didn't do that, because that's actually more complication on the client side that we didn't have to deal with when we just run the agents. Plus, running an agents have some other benefits for us.

Then what collector does is in a very simple case, it just accepts all these spans being sent by old applications, or in our case, by all the agents, and stores them to the database. Jaeger supports not just Cassandra. You can run Elasticsearch out of the box from Jaeger. Recently, we've implemented the plugin framework, where people can implement storage support for other storage types, like InfluxDB was recently implemented, and Couchbase I think is still in development. But Cassandra and Elasticsearch has supported out of the box.

Then a third part of Jaeger's sort of backend is Jaeger query service, which is another microservice which serves the frontend JavaScript-based, React-based frontend, and then it tells who does – Reads from the database and does a translation. So this is like a very minimal setup that you need for Jaeger. Internally, we also have other things. We have a data pipeline where spans [inaudible 00:37:04] will send to Kafka. On Kafka, we run various fringe jobs that perform aggregations, like building the service dependency graphs, building tracing quality metrics for people and all kind of other interesting stuff that we're still working on.

Yeah, there are sometimes even more complications in the Jaeger infrastructure. You could actually put Kafka in the middle between collector and the ingester, which is what we're using at Ubers. It was recently released as open source as well. But that's for – It gives like various other benefits. It's obviously complicates the deployment, but it gives the benefits of a more flexible system in terms of if you get like traffic spikes.

**[00:37:45] JM**: Okay. Let's zoom in on these different components a little bit. So on every service, if I understand correctly, like whether on the Uber Eat service or the pricing data service, I have – If I want distributed tracing, I've got Jaeger client and Jaeger agent. Is that correct?

**[00:38:03] YS**: So, Jaeger client is a library that runs inside this service process, and Jaeger agent is a host agent. It like runs once per host.

**[00:38:11] JM**: Oh, okay. So this multiple services deployed to each host, because you've got multiple containers on each host?

**[00:38:19] YS**: Correct. Yes. The Mesos can schedule basically multiple workloads on a given host.

**[00:38:25] JM**: Got it. So the Jaeger client is a per container module. Then each of those containers is forwarding their particular services information to the Jaeger agent, which is a host level system.

**[00:38:40] YS**: Mm-hmm.

**[00:38:41] JM**: Okay. Then the Jaeger agent is taking all of that client information and forwarding it to the collector. Then the collector is just storing it in Cassandra.

**[00:38:54] YS**: Correct.

**[00:38:54] JM**: Okay. So, what did you learn from Uber's earlier distributed tracing work that you put into the Jaeger project?

**[00:39:05] YS**: It's a good question, whether there's a lot of learning – I think one thing – I wouldn't call it so much learning, because we decided to do that by design upfront. But the way we designed communication between clients and the backend was bidirectional, which is not how many other tracing libraries are implemented including like open sensors and open telemetry.

So what we had was a feedback loop. The reason we did that is because one of the challenges with tracing is that we cannot trace every single request in production, because we're going to generate so much data that's probably going to be more network traffic than the actual business traffic.

So most tracing systems in production today employs some sort of sampling, and we sample pretty heavily actually at Uber, but I can't give a specific number because that's – The reason for the feedback, well, we implemented so-called adaptive sampling, where a sampling decision is made somewhere in the client at the very first span of a trace where the trace ID is first time created, and that decision is propagated with a trace ID through the rest of the call graph so that

every service knows whether that request needs to be like traced and the data needs to be collected or not.

I mean, I think we started originally with maybe one in a thousand probability of these sampling decisions. But eventually we evolved into adapting sampling, because with a fixed probability, the challenge that we had was if you take any service, each service in general exposes typically more than one endpoint. Those endpoints very often have very different QPS, like query per second, or like a volume of traffic.

For example, you can have – I don't know, your read requests maybe 10 times more frequent than your write requests. Something like that. So if you're using a single probability for that service, then you're getting much of your traces from your low QPS endpoints. If the probability is very small, like one in a thousand or even less, then potentially you're not getting any data whatsoever from certain endpoints, and that's kind of a bummer, because you want to see what happens in your architecture, how all the services connect, what the dependencies between services are.

So we didn't like that. So we've implemented a different type of sampling, where a probability was not per whole service or per whole organization, but actually per service, endpoint. That was great, because I can have one endpoint being traced with 100% because it's like so rare that we can capture all the data. Whereas as a high QPS endpoint is going to be traced with a very low probability. We're still getting a lot of data about it.

But once you move to that mode and if you start counting how many services and endpoints actually exist in Uber, so 3,000 microservices times – I don't know, on average maybe 10 endpoints. There're a lot of probabilities. No one can manage that manually. So we've built a system which manages those probabilities automatically by setting a certain target of a rate of tracing we want to see from a given – For many things, essentially. Then calculating what the appropriate probability for that service given the traffic that we're seeing from that service.

Then that system also allows us to deal with traffic spikes if suddenly you deploy some new function and you start receiving twice as many requests. That functionality required us to build client libraries in a slightly different way than most of the open source client time, the libraries at

the time, where they were having this feedback loop where the probability were constantly recalculated no a backend and flowing through collector to age and back to the clients. So that was I think one of the big changes that we had to make to our client space.

[SPONSOR MESSAGE]

**[00:43:01] JM**: Today's episode of Software Engineering Daily is sponsored by Datadog, a monitoring platform for cloud scale infrastructure and applications. Datadog provides dashboarding, alerting, application performance monitoring and log management in one tightly integrated platform so you can get end-to-end visibility quickly, and it integrates seamlessly with AWS so you can start monitoring EC2, RDS, ECS and all of your other AWS services in minutes. Visualize key metrics, set alerts to identity anomalies and collaborate with your team to troubleshoot and fix issues fast.

Try it yourself by starting a free 14-day trial today. Listeners of this podcast will also receive a free Datadog t-shirt. Go to softwareengineeringdaily.com/datadog to get that t-shirt. That's softwareengineeringdaily.com/datadog.

[INTERVIEW CONTINUED]

**[00:44:06] JM**: The user of a distributed tracing frontend is probably – By frontend, I mean, the dashboard where I'm looking at my traces and I'm trying to find out what are the problems with my distributed system or I'm trying to just improve the latencies of my distributed tracing. This user needs to be able to search through this distributed traces or aggregate these distributed traces. Describe what the goals of – When I sit down in front of my distributed tracing frontend, what am I trying to accomplish and what do I want my interface to do?

**[00:44:46] YS**: It's a very good and interesting question, because I gave recently a whole number of talks on this showing how – You mentioned latency and when you were describing overall what tracing, how it works, the spans and the call graph, you also mentioned how long things took.

This is kind of an interesting aspect, because a lot of people are actually thinking of tracing as a performance optimization, sort of application performance management tool where you are

worried about latency and you want to find things which are slow. That's certainly a very valid use case with tracing.

But what I've noticed, like in my experience at Uber, is that that use case never actually got picked up by Uber engineers as the primary use for tracing. I don't know if I know the reasons for that. I can speculate. I think one of the reasons is probably like Uber is still very young and very fast-growing system. So that we have like a bigger fish to fry than like worrying about latency. I mean, in some cases, where it's like very bad. Obviously, people worry about it.

But there are other aspects of the microservices architecture that tracing isn't just more useful for us. Specifically, that aspect is a complexity, because the other thing that you want to use tracing for is really for troubleshooting issues in production. Those issues may not necessarily be latency related. In fact, I think it's a kind of a known fact in the industry is that the majority of production outages happen because of the change management, because you release a new version or release a new configuration and something just didn't work as a result.

Sometimes that work aspect may manifest in the latency, but it can also manifest in all kinds of other ways. It maybe you are getting bad message format and you're not able to parse it. So the user restores in there, something like that, or maybe some other service has just gone somewhere and you can't even reach it.

I mean, it's impossible to enumerate all the possible problems that could happen. But that's just kind of the primary useful tracing at Uber is trying to understand, "Okay. Let's say we have a business metrics. How many people are taking trips in New York?" We have a sort of an average number during the day week over week. Suddenly that metrics drops in half. That's obviously a business outage. We need to fix it immediately.

How do you go from there to – And you get an alert about it saying, "Yes, your metrics dropped." But how do you go from there to actually finding out what in your 3,000 microservices architecture is actually broken? You can look at the history of changes that just went out to production, but given the number of services in teams, those are going to be like thousands of events in the last hour or so. There are lots of changes going on all the time. So that doesn't help.

So tracing is really kind of the only tool that helps us to quickly navigate this complex system and find – If not find the actual root cause, but at least pinpoint where the problem is occurring. That is sort often more challenging task finding where the problem is than deciding what the problem is and how to fix it. So like navigating to the issue is harder. That's the second use of the distributed tracing that got a lot more usage and a lot more benefits to Uber than purely, "Oh, I'm going to use tracing for performance optimization." We obviously have both. There're some power users who did optimize their services for latency, but this is not a prevalent use case.

So, from UI perspective, is that you want a system that helps you with either of those. The classic, the Gantt chart view that tracing systems provide, like Jaeger does that, Zipkin does it. Essentially, every iterating system gives you by default the Gantt chart view of the trace. That is very helpful for latency investigation, because, well, you can kind of collapse everything and see what the longest parts. You can do critical path analysis and understand certain things.

There are also – I know you had discussions with Ben Sigelman on your show previously. So Lightstep is building way more advanced tools to actually investigate latency, because sometimes even with latency, even though you can see where the latency happens in a single trace, you may not be able to explain why that happens, because there might be some contention on a resource held by some other transactions. So you need to go into more aggregated processing of the traces to figure out, "Oh! What's the correlated transactions or different traces that heat that resource and why they're containing on it?"

So, even for latency, you kind of need usually more than one trace view. For outage resolution, it depends on how you detect the outage. So one way is you can also present the user with aggregated view of your system behavior saying, "Maybe if they know that the specific business metric that fired an alert is tied to a specific endpoint on your frontend API service." Let's say riders – I don't know [inaudible 00:49:49] or maybe put a destination on a map, and that thing is not working, right? So, you might be able to figure out what endpoint is responsible for that.

So, you can go and build an aggregation and plot, graph a service map for that endpoint. What are all the dependencies, and overlay certain metrics information on top of that saying, "Well, all

the requests within the normal latency, within the normal error rates. You can color code them with like whatever, the red of green," and that potentially can lead you to a problem where like which service is actually is the source of the problem or breaking the whole flow.

There is another way which is what – We haven't gone into that first approach yet, because it actually allows – It requires a lot of standardization for us to gather all the data. We're not there yet. But we went into a different approach where you can also detect these types of outages by doing synthetic probing from some service, like which may sit outside of your data centers and pretend to be taking trips.

That service, it's not going to do like a huge volume of request similar to production traffic, but it's a period request that it sends and says, "Oh! I know that to take a trip, I need to execute the steps. Well, I see that is not working somewhere." That actually gives us a single trace, because as soon as that thing fails in that synthetic prober, it can give us a trace ID and then you can go into the UI and investigate what specifically in that trace ID is failing. So that's a different way that you may approach sort of using tracing to troubleshoot production issues.

**[00:51:29] JM**: Well, there's so much there that we could dive into. But we are a little up against time and there's a totally different topic I wanted to ask you about assuming you're cool talking about it. You were in finance for many years including 2008 working as an engineer at financial companies that really endured the 2008 financial crisis. I'm just wondering what it was like to live through 2008 in the middle of the storm.

**[00:52:01] YS**: That was not the favorite years certainly. I was working at Morgan Stanley, which was across the street from Lehman Brothers. So it was fun one day when we come to work and like the building, Lehman Brothers building, it is all surrounded by new structs, because they just went down.

Then during that same week, Morgan Stanley stock dipped to like from $20 to $5, and we weren't sure. There was a movie about this, which I really liked, where they were showing how all these was unfolding as well. Because I didn't know that, the rest of the industry story, right? I only knew like from what's happening from the Morgan Stanley. So, there's a lot of worrying that the whole company might go down because of that.

Eventually, yeah, once this bailout happened and sort of the stock price stabilized a bit. I think after that there was like a lot of new work just came down the pipe with the regulations, with [inaudible 00:53:03] and all kinds of – What they call them, the fire drills, where you have to prove that you can withstand sort of fluctuations in the market.

I remember a story, someone said like, "If interest rates go –" This is like look at one single trade for derivatives. It was like in – I don't know, hundred million or billion notional amount, and then if the interest rates industry go up by 1%, the whole company goes down because of this one single trait. That was kind of an environment which is very weird. So it was worrying times, but it passed.

**[00:53:39] JM**: So, I worked very briefly in the trading world. This is back in 2014. But I personally kind of liked moving from the finance world into kind of the product world. Not that finance is not a product, but I just remember looking at Hacker News every day when I was working on the trading company and seeing people building these SaaS companies. I was just like, "This is looks – I would rather be doing this." But I don't know. What's your experience like comparing working in finance where you're kind of just like playing a big poker game versus the kind of SaaS company world?

**[00:54:17] YS**: I kind of had the similar feelings that I thought that there's a lot of cool technological innovation and work that's happening outside of finance. I myself was on what you call product side. So I wasn't really even working on the infrastructure problems at Morgan Stanley where I've spent most of my time. It was building a trading system, trade capture, processing frontend, all of these things. At some point, it became kind of worrying, because you were not solving technologies problems. You were solving more of business process problems. There is obviously some challenges there, but it kind of became repetitive to me. I wanted really to dive in into more of a distributed systems, infrastructure work.

So that's why I started looking for something else, and definitely working at Uber was way more fun not just because Uber was a startup. I think if I went to another like more stable, like Facebook or Google. But if I worked on the infrastructure team, I would had as much fun there

as well. It's just like that domain is much more interesting to me than the product side of the businesses.

**[00:55:27] JM**: Yuri, thank you for coming on Software Engineering Daily. It's been really fun talking.

**[00:55:32] YS**: Thank you very much for having me. It was great.

[END OF INTERVIEW]

**[00:55:38] JM**: Commercial open source software businesses build their business model an open source software project. Software businesses built around open source software operate differently than those built around proprietary software.

The Open Core Summit is a conference before commercial open source software. If you are building a business around open source software, check out the Open Core Summit, September 19th and 20th at the Palace of Fine Arts in San Francisco. Go to opencoresummit.com to register.

At Open Core Summit, we'll discuss the engineering, business strategy and investment landscape of commercial open source software businesses. Speakers will include people from HashiCorp, GitLab, Confluent, MongoDB and Docker. I will be emceeing the event, and I'm hoping to do some on-stage podcast-style dialogues.

I am excited about the Open Core Summit, because open source software is the future. Most businesses don't gain that much by having their software be proprietary. As it becomes easier to build secure software, there will be even fewer reasons not to open source your code.

I love commercial open source businesses because there are so many interesting technical problems. You got governance issues. You got a strange business model. I am looking forward to exploring these curiosities at the Open Core Summit, and I hope to see you there. If you want to attend, check out opencoresummit.com. The conference is September 19th and 20th in San Francisco.

Open source is changing the world of software and it's changing the world that we live in. Check out the Open Core Summit by going to opencoresummit.com.

[END]