EPISODE 304

[INTRODUCTION]

[0:00:00.3] JM: Most tech companies are moving towards a highly distributed Microservices Architecture. In this architecture, services are decoupled from each other and communicate with a common service language, often JSON over HTTP. This provides some standardization, but these companies are finding that more standardization would come in handy under certain scenarios.

At the ridesharing company, Lyft, every internal service runs a tool called Envoy. Envoy is a service proxy. Whenever a service sends or receives a request, that request goes through Envoy before meeting its destination. Matt Klein started Envoy and he joins the show to explain why it is useful to have this layer of standardization between services. He also gives some historical context about why Envoy was so helpful to Lyft.

I really enjoyed this episode and I also enjoyed the follow up to this episode, which was me seeing Matt Klein at the Microservices Practitioner Summit. I think the talks for the Microservices Practitioner Summit will be up by the time I release this episode. If not, I recommend checking out Matt Klein's other talks.

[SPONSOR MESSAGE]

[0:01:16.3] JM: Life is too short to have a job that you don't enjoy. If you don't like your job, go to hired.com/sedaily. Hired makes finding a new job enjoyable, and Hired will connect you with a talent advocate that will walk you through the process of finding a better job. It's like a personal concierge for finding a job. Maybe you want more flexible hours, or more money, or remote work. Maybe you want to work at Facebook, or Uber, or Stripe, or some of the other top companies that are desperately looking for engineers on Hired.

You deserve a job that you enjoy, because you're someone who spends their spare time listening to a software engineering podcast. Clearly, you're passionate about software, so it's definitely possible to find a job that you enjoy. Check out hired.com/sedaily to get a special offer

for Software Engineering Daily listeners. A $1,000 signing bonus from Hired when you find that great job that gives you respect and salary that you deserve as a great engineer. I love Hired because it puts more power in the hands of engineers.

Go to hired.com/sedaily to get advantage of that special offer. Thanks to Hired for being a continued longtime sponsor of Software Engineering Daily.

[INTERVIEW]

[0:02:42.3] JM: Matt Klein is an engineer at Lyft. Matt, welcome to Software Engineering Daily.

[0:02:47.0] MK: Thank you very much.

[0:02:48.7] JM: The place to start this conversation is with a discussion of how a modern service-oriented architecture, or Microservices Architecture, looks, because we're going to be talking about Envoy, and we'll get into what Envoy is. For some of the common challenges of service-oriented architecture, just give an overview for how people are building service-oriented applications today.

[0:03:13.0] MK: Sure. Yeah, so we've come a long way, probably, over the last 5 to 10 years, and what you're seeing today is you have lots of companies that tend to be kind of what we call polyglots, so they'll have multiple languages. It's quite common now where people might have PHP, and Python, and Go, and Java, Scala, and kind of all these different languages. Typically, what we see today is that people start to move in their monolithic architecture, that's typically how people start. They'll have, essentially, one application, they'll have one back-end database, and then their clients will effectively kind of call that one service.

Then, as organizations tend to mature, what ends up happening is people start building different services, typically, in multiple different languages. They'll typically use different, I would say, language-specific libraries. For example, in PHP, they might be using cURL to call service calls. In Go they might be using a Go-specific library, etcetera.

For, I would say, all but the most sophisticated companies, what ends up happening is people essentially use these libraries to make these service calls and then they'll often have different problems, and those problems might be related to networking failures, or they might be related to partial implementations of things like not doing circuit breaking, or not doing things like retry, things like that. Then of course, you also start having a bunch of different problems around how do you actually route request? How do you know what back-end service to call?

The way that most architectures look today is you have a bunch of services, you typically have some type of load balancer, whether that's kind of service side kind of application library, or it might be some kind of middle load balancer, like a hardware F5 or an Amazon elastic load balancer. Then you're essentially stringing these calls together and you're attempting to figure out what's going on. The way that most people do that is they have both logging, they have stats, and if they're particularly sophisticated, they might also have tracing.

Because everyone is doing this in different ways, and using different load balancers, and different languages, and different libraries, it's quite common that people have kind of partial implementations of all of these things. They might have some logging, but they might not have stats, and very few people have tracing. Or one of their languages might have sophisticated retry support and circuit breaking, whereas other languages don't. That's kind of the current lay of the land, effectively.

[0:05:53.4] JM: What you're describing is this wide heterogeneity of databases, programming languages, communication protocols, cloud platforms, selections of tools to use in those cloud platforms. Within whatever subset of tools that you choose, you've also got a diverse buffet of different observability tools that you could use. You end up within a big company as this wide range, there's no standardization. This difficulty of observability that comes out of all these diversity — To put a finer point on this, explain the issues that a big company will have to deal with in terms of debugging and, I guess, just day-to-day understanding of what's going on in the architecture that this diversity yields.

[0:06:48.1] MK: Yeah. When I talk to people about service-oriented architectures, I typically say that observability is probably the most important thing. I think, historically, people have focused a lot on different features and obviously it's quite portent in terms of what different protocols

there are and whether you can do retries, and whether you can do circuit breaking and different types of load balancing.

At the end of the day, when people build these architectures, it's important to know both what's going on, and when things go wrong, you have to be able to quickly identify kind of where the problem is. If you look at these architectures, and as you were saying before, it's kind of this Wild West, right? You have containers as a service, you have infrastructure as a service, you have on premise, you have five different types of load balances, let alone, different vendors, different databases.

As people build out these architectures, it can be extremely difficult to understand when things go wrong, where they're going wrong, and then to enable kind of developers who, in our modern systems, are also the operation's people, to figure out how to fix that, can be incredibly difficult.

To kind of go into what are the problems, if you look at a kind of modern architecture that might be built on something like Amazon web services, there could be real hardware failure, there could be real network failure, there could be virtual kind of hardware failure, or virtual network failure. The application could crash. There could be a garbage collection event. There could be buffers that basically fill and don't fill. At that point, it can become incredibly difficult to actually understand what's going on.

Being able to understand and dig into at the six, or seven, or eight different levels of the stack, what component is actually causing the problem, so that you can debug and then hope to rectify it. That's probably the most important thing that people need to do in these modern architectures.

[0:08:51.1] JM: Moving to the project that you started called Envoy, the mission statement of Envoy is that the network should be transparent to applications. When network and applications problems do occur, it should be easy to determine the source of the problem. That's an ambitious mission statement, because there's so much involved in a network and application stack as we have just described. What does that actually mean in practice?

[0:09:22.6] MK: Sure. What we've attempted to do with Envoy is kind of — As I was saying before, you've got this situation where people have different libraries that typically make service calls and kind of allow people to string together their kind of service-oriented architecture. What you'll find is that at very few companies, if you're at Google, or if you're in a place like Amazon, or kind of other large companies, they have the resources to focus on a couple of language. They'll typically have invested a huge amount of engineering resources in making client-side libraries that enable people to kind of operate in this so-called Wild West.

If you're Twitter, or you're using a library called Finagle, right? If you're at Google, you're using a library called Stubby. What these libraries do is, in one place, they allow application developers to get access to a super rich set of features both in terms of networking. Again, things like circuit breaking, and retries, and rate limiting, and buffering, and kind of all of those things. Also, distributed tracing, stats, and also logging.

If you're not at one of those companies, you often have partial implementations of all of these things. Having observed kind of what people do if you're at a big company that is lucky enough to have these robust libraries, or if you're at a small company where you're kind of using a mishmash of different things, Envoy takes the so-called sidecar approach, and what Envoy does is it's s self-contained server. It's a proxy that you run alongside your application.

The idea behind Envoy is that an Envoy proxy node is collocated with every application node, and the application, essentially, only talks to Envoy. The application makes outgoing or egress calls through Envoy, and the applications receives incoming or ingress calls through Envoy.

From the applications perspective, all it ever talks to is its local Envoy which is sitting on local host. Then, contained within this Envoy proxy is a rich set of service discovery features of figuring out who there is to talk to, health checking, figuring out which of the upstream nodes are actually healthy, routing, load balancing, circuit breaking, rate limiting kind of all of these features. Then, on top of that, very, very rich capabilities to do logging, to do tracing, and also to commit stats.

What you have is you build — Using the sidecar approach, you build this service mesh. In this mesh, the application traffic basically transits the mesh, but from the application perspective, it's not really aware of any of the aspects of kind of the underlying networking architecture.

What that ends up meaning a practice is something that's pretty incredible, where if you're writing an application, and to use Lyft as an example, let's say that we have a service called user service which provides user information, or service call location service, which provides location information, you have a very thin client in your application code which says, "I want to talk to the location service," and that thin client knows how to talk to the local Envoy. The Envoy will figure out which upstream node to actually send that request to. It will annotate it. It will instrument it. It will enforce timeouts. It will do all of these different things. Then, ultimately, it will return that response back to the calling application. That same code works whether you're at Dev, or you're in staging, or you're actually in production. The application developers, they write their code once, they understand kind of what the underlying topology actually looks like and they just essentially make their service calls.

[0:13:24.7] JM: Now, I want to just reiterate a little bit to make sure I understand it properly, because it took me a while studying this to kind of understand exactly what it's doing. On every host that is running a service at Lyft, for example, you have a separate sidecar cross. You call this a sidecar where you just have Envoy sitting there running, and whenever the service would want — Whenever the service running on the host want to make a call to another service, you first talk to Envoy, and then Envoy ferries that request to the next service, and that is the process of proxying. That when we talk about service proxying, that's what we're talking about, the service is not contacting any other service directly, it's calling Envoy, and then Envoy goes and talks to the host on another — The host of another service, and then it's talking to the Envoy process on that service which will then the Envoy process on that other service will then talk to the service itself. Is that correct?

[0:14:28.0] MK: Yeah, that's 100% correct. That might sound, I would say, counterintuitive on several levels. Let me kind of explain a little bit. On the first level, some people might hear that and they might ask, "Well, it seems pretty inefficient that you're going to proxy this call twice and have to go through these two proxies just to get back to your application." When you start thinking about all of the functionality that gets implemented within Envoy, that then is accessible

from any language. It doesn't matter what language or what runtime you're using, whether it's PHP, or Go, or Scala, or Ruby, or Python, name your language, it just works. You start getting a better picture of in the long run, it is substantially more efficient to have correcting robust implementations of all of these primitives, than to have to re-implement them in every single application.

From an observability standpoint, it becomes even more critical, because you want a consistent set of logs, and stats, and also tracing, and those, of course, possible to implement kind of consistent observability output in every individual application language. It's an incredible development effort to actually have that happen.

From our perspective and what we've seen at Lyft, though it sounds counterintuitive to double proxy all of these requests, that double proxy allows us to have this trusted mesh infrastructure. It gives us not only huge amount of features, but it also gives us this very rich observability output, which is pretty incredible from an operations perspective.

[SPONSOR BREAK]

[0:16:18.5] JM: Simplify continuous delivery with GoCD, the on-premise, open source, continuous delivery tool by ThoughtWorks. With GoCD, you can easily model complex deployment workflows using pipelines, and you can visualize them end to end with its value stream map. You get complete visibility into and control of your company's deployments.

At gocd.io/sedaily, you can find out how to bring continuous delivery to your teams. Say goodbye to deployment panic and hello to consistent predictable deliveries. Visit gocd.io/sedaily to learn more about GoCD.

Commercial support and enterprise add-ons, including disaster recovery, are available. Thank you to GoCD and thank you to ThoughtWorks. I'm a huge fan of ThoughtWorks and their products including GoCD, and we're fans of continuous delivery. Check out gocd.io.sedaily.

[INTERVIEW CONTINUED]

[0:17:29.5] JM: Yeah, and I can speak to like from my person experience, I worked at Amazon briefly, and when you start at a company like Lyft of Amazon, from day one, you got to hit the ground running. If you're working on some new service, you don't want to have to think about stuff that every service has to think about. Rather than, "Okay, now I've got to write a health check for my service," "Okay, now I've got to implement a circuit breaker in my service," "Okay, now I've got to implement rate limiting in my service," you need all these things taken care of for you.

You might as well have all these things taken care of for you, not only because it saves time, but also because, as you said, you need to deliver a certain SLA to all the other service level agreement, to all the other services in the company, because most of these — I don't know if Lyft operates this way, but a company like Amazon, or a company like Twilio. Every other service in the company can call you if they want to, and you have to be able to guarantee certain statistics, like how quickly you're to respond. The only way you can signal to other people that you're healthy is by having a reliable source of truth. That is why it makes sense to have this standardization point between every service.

[0:18:51.8] MK: Yeah, that's 100% correct, and I would add to that a little bit, which is to say that once you've decided that you're going to invest kind of this substrate that does these various things, performance does end up being very important. I don't necessarily mean performance in terms of total throughput, though that's very important. For most companies, if they're not a Google, or a Facebook, or a Microsoft, or an Amazon, developer time tends to be more expensive than actual infra-cost, right?

Look, what I think people discount a lot, particularly from some of these service substrate work, is that where a lot of people end up spending their debugging time, is out at the tail in terms tail latency. As you're saying, if you're trying to do this kind of SLAing — Typically, SLAs are defined at P99, or P99.9. It's very important to have kind of reliable stats out at that tail. What you're seeing now is you're seeing kind of different, I would say, implementations that are popping up that are along the lines of what Envoy is doing.

Envoy is written in C++ for a reason, not because we think that everything should be written in C ++. That's obviously not the case, but for something at that kind of critical infrastructure level,

having something that's written kind of in that native code, it doesn't use garbage collection, it doesn't add these P99.9 pauses. It tends to be very important, because you kind of don't want this observer effect —They talk about observer effect in physics, where if the act of observing, basically, changes what you see, then you don't actually know what you're seeing.

It becomes really important to have this component in your system that has, I would say, very reliable performance, because that gives you kind of some of the observability that you're going to kind of pin all of your SLAs around kind of all of your experience.

[0:20:59.7] JM: Let's talk a little bit more about what the features that you need out of a service proxy, because as we said, Envoy is a service proxy, and it's built on the idea of being a TCP L3, L4 proxy. I don't know exactly what that means, so if maybe you could explain it, or maybe it's not worth getting into. Why don't you just —

[0:21:20.2] MK: No, it's fine. Yeah. Sure.

[0:21:21.7] JM: Okay. Sure. In any case, I just wanted to start talking about what are the features of Envoy.

[0:21:26.7] MK: Yeah, sure. From an L3, L4 perspective, by that, I just mean that kind of at its core. Envoy is kind of a TCP / IP proxy. We don't currently support UDP, but we probably will in the future. The idea is that at its core, it's basically — Bytes come in, those bytes might get processed in some way, we're going to pick some place to send those bytes and then we're basically going to forward those bytes.

By itself, that doesn't necessarily sound, I would say, very interesting, but when you start coupling the byte processing with filters. Basically, filters can go ahead and they can look at these bytes and they can do things. You can start composing very interesting systems on top of that. At the base level, you can just do a raw TCP proxy, which basically says, "I'm going to do some SSL bridging, I'm going to connect some source, and I'm going to load balance these connections all to some back-end." Basically, a very simple raw TCP kind of HA proxy like use case.

Then, because of this byte processing filtering system, you can start doing other protocols. Out of the box, we actually support MongoDB-sniffing. At Lyft, we use MongoDB quite a bit. We actually use Envoy for all of our MongoDB connections. We do that for rate limiting purposes, we do that both for actually getting stats from MongoDB. We actually sniff that database communication as it goes back and forth and actually generate all of these interesting stats.

Then, the most obvious case of this proxy is people want to do what we call Layer 7, or application proxying, typically using HTTP. Obviously, in modern architectures, whether it's normal REST calls or kind of new GRPC-based calls, I would say those kind of those protocols are fundamental underpinning of kind of most service-oriented architectures.

Envoy has very sophisticated kind of handling for GRPC as well as kind of normal REST calls. We could do filtering at that layer also. We can look at individual request and responses, and we can look at headers, and we can look at body data, and we can start implementing things, like routing, buffering, we have some pretty cool features around GRPC bridging. We kind of operate at different layers of stack. There's the kind of the TCP bytes in, bytes out processing, and then we have also kind of very sophisticated handling of higher level protocols, whether it'd be MongoDB, we've got Redis that's actually coming, we edit HTTP. There's kind of a lot of different processing that's happening there.

[0:24:19.4] JM: You mentioned GRPC. GRPC — What layer — Where in the stack exactly does GRP in terms of — I guess GRPC is something that is over HTTP, right? That would be a subset of the HTTP calls that you're talking about.

[0:24:35.6] MK: Yeah. GRPC is actually kind of interesting. For those that don't know, GRPC is relatively new. It probably came out in the last couple of years. It's from Google. Kind of the idea behind GRPC is that you want to more strongly define your service calls. Service calls are defined using an IDL, that's Interface Definition Language, and they use an IDL language called Protobuf. Protobuf has been around for a super long time. It's kind of related to Thrift from Facebook.

Protobuf was an open sourced by Google, probably 7, 8, 9, 10 years ago. What GRPC does is it takes a Protobuf definition and it uses for framing kind of a higher level protocol HTTP 2. When

you want to multiplex different requests, you basically use that HTTP 2 protocol and you send these Protobuf messages inside of it. You can think of it as similar to REST, but instead of using something like JSON, which isn't strongly typed. It's kind of a request response framework, which ends up being strongly typed.

[0:25:44.4] JM: Got it. Now, there's also the aspect of proxying between, I guess, HTTP 1 and HTTP 2. Let's assume I don't know anything about the different between HTTP 1 and HTTP 2. Why do I want to proxy between these two?

[0:26:01.1] MK: Right. There's obviously two different protocols, and the two protocols are kind of interesting, because you have the second version of the protocol which it's not pretty widely supported. I think it was probably ratified maybe three or four years ago and it's based on Speedy. Obviously, you have the first version of the protocol which has been around for 10, 15 years or something like that.

From a semantic perspective, both versions of the protocol — They essentially have the same features. For example, if you're familiar with kind of old style REST calls or just kind of web calls, you're going to know that a standard call is going to have headers, it's going to have some body, it's going to have like get post and kind of all of those different things.

The second version of the protocol, it doesn't change any of the features. You still effectively have headers, and you've got body, and you've got trailer, and you have different things, but it changes it from a text-based protocol to a binary-based protocol.

The most important features is it allows you to do what we call multiplexing. Over a single connection, you can send multiple requests. That's a big difference from HTTP 1, where with HTTP 1, you could do keep-alive, but if you wanted to send five requests the same time, you have to have five different TCP connections, and that ends up getting fairly inefficient. Whereas with the second version of the protocol, you can have one TCP connection and you can actually multiplex these requests and responses, or what we call streams over this single connection.

The binary protocol ends up being a lot more efficient, and then you have this multiplexing which is also a lot more efficient. From a semantic perspective, there isn't a lot of difference. You still say, "Get slash," or, "post foo," or something along those lines.

[0:27:54.0] JM: Great. We have touched on the granular aspects of Envoy. Let's get into some of the features that you'll get out of it. Let's start with service discovery. You'd already mentioned service discovery. We've explored the basics of service discovery on previous episodes. How does Envoy look at service discovery?

[0:28:12.6] MK: Yeah, just for like a general recap. Service discovery is obviously the process by which you identify one of the members of your service that you can end up calling. I would say Envoy does something that I think is fairly unique amongst these systems where, historically, people have treated service discovery as a fully consistent problem. By that, I mean they end up using data stores, whether at ZooKeeper, whether it's Etcd, or whether it's Consul, that use kind of Paxos-like algorithms to have a fully consistent view of what nodes are essentially currently in your active set.

There's lots of companies that do this and they do it at large scale. One of the things that we kind of decided when we were initially designing Envoy is there tends to be a lot of pain in terms of operating those fully consistent kind of service discovery systems, just because operating ZooKeeper and operating Etcd tend to be fairly, fairly fragile.

As you probably know, as in most things, if you can take a problem that you're typically doing full consistency for and you can make it eventually consistent, it ends up being a lot easier. One of the things that we've realized with Envoy and kind of that it made an explicit point of, is realizing that from a service oriented architecture perspective, service discovery is actually eventually consistent.

Nodes are coming and going, you're doing health checking. It doesn't need to be that every node in your system has a fully consistent view of what the upstream hosts are, mainly because overtime, as long as that view converges, it doesn't matter if it's out of date by 500 milliseconds, or sometimes even 10 seconds, or even 30 seconds, because each of these nodes from a fanned-out perspective are effectively looking at their own view of the system.

Envoy, we implemented this reference implementation of our own service discovery, and it's a few hundred lines of Python that's basically built on top of DynamoDB. If you look at the complexity of that system versus something like ZooKeeper, or Consul, we wrote that code probably a year or over a year ago now and we essentially haven't touched it. We honestly never touched it, it just runs.

We kind of take this idea where at every layer, this systems is eventually consistent. How it works at Lyft is when every machine comes up, it literally has a Cron job that runs once per minute, and it says, "Hi, discovery service. Here I am. Here's my IP address. Here's my node type." That information is written into Dynamo in an eventually consistent way. We use eventually consistent reads to actually readout our service discovery members, those are cached in a memory within the service, and then each Envoy talks to the discovery service every 30 to 60 seconds [inaudible] and it basically gets the correct node numbers and then it does health checking.

If you look at it end to end, it could be anywhere from probably one to three minutes before every Envoy essentially knows about kind of the addition, or loss of a particular node. What we find in practice is that though kind of the system is not fully consistent, it's extremely reliable, because this backing store which tends to cause a lot of people problems, if they're basically relying on ZooKeeper to be up to know which nodes to talk to, and then ZooKeeper goes down, that can be a frequent cause of kind of large scale failure in these systems. We kind of made that explicit goal to kind of go this eventually consistent system.

[SPONSOR BREAK]

[0:32:23.9] JM: You are building a data-intensive application. Maybe it involves data visualization, a recommendation engine, or multiple data sources. These applications often require data warehousing, Glucode, lots of iteration, and lots of frustration.

The Exaptive Studio is a rapid application development studio optimized for data projects. It minimizes the code required to build data-rich web applications and maximizes your time spent

on your expertise. Go to exaptive.com/sedaily to get a free account today, that's exaptive.com/sedaily.

The Exaptive Studio provides a visual environment for using back-end, algorithmic, and front-end components. Use the open source technologies you already use, but without having to modify the code, unless you want to, of course. Access a k-means clustering algorithm without knowing R, or use complex visualizations even if you don't know D3.

Spend your energy on the part that you know well and less time on the other stuff. Build faster and create better. Go to exaptive.com/sedaily for a free account. Thanks to Exaptive for being a new sponsor of Software Engineering Daily. It's a pleasure to have you onboard as a new sponsor.

[INTERVIEW CONTINUED]

[0:33:54.9] JM: That makes complete sense to me. Just to give people a little more of a lesson in kind of the eventual consistent versus consistent in practice, can you talk about how the — Or whatever you want, describe ZooKeeper/Etcd/the eventually consistent systems, how the problems of those systems manifest if you use those for service discover. How do those lead to large scale failures, and how does that, in contrast, get minimized in an eventually consistent service discovery system?

[0:34:25.8] MK: Yeah. Let's just talk about something like ZooKeeper, but in practice, within reason, it all looks similar whether it's Etcd, or Consul, or something like ZooKeeper. These are all what we call fully consistent systems, where when you do a write to it, it has to undergo a fairly complicated algorithm to actually figure out who's the master, it has to handle failovers, it has to do really, really, really complicated logic that people spend years and years and years working on to kind of make sure that it gets right.

The way that people have historically used systems like ZooKeeper is each node will basically take a lease into ZooKeeper, and each node will say, "Hey, here I am. I'm going to tell ZooKeeper that I'm here," and then all of the other nodes will basically watch ZooKeeper. You could think of it, it's almost like a directory tree of files, right? You have a file called /service /foo,

and then in that directory are basically all of the members. Each member is kind of touching their file into that director tree and then all the other people are essentially watching that tree. Though that sounds really simple in practice, there's all of these really complicated algorithm magic that's happening behind the scenes to have that work.

The problem is that if the ZooKeeper cluster, which is a very complicated piece of code, with masters and slaves, if it goes down and you're processing is essentially very naïve, where you essentially rely on that cluster being up, your entire system is down, because every node now has no idea who to talk to. You're basically hard down.

Over the years, what people have done, and we actually saw this back at Twitter, where I would say four or five years ago we would have, not regular, but somewhat regular outages basically related to ZooKeeper. They started building kind of eventually consistent, I would say, behavior into the libraries. Finagle wouldn't necessarily watch ZooKeeper directly, it would keep a watch, but then if it failed, it would kind of use the last state, so it would keep that state within its own memory. Then, when ZooKeeper came back, it would basically refresh its state.

What's kind of interesting about that is that from that perspective, and this is what kind of a lot of people end up doing, is they take this fully consistent system and then they basically make it eventually consistent. What we've set from the get go is that it's pointless to take a fully consistent system and make it eventually consistent. Why don't we just make it eventually consistent from the get go?

[0:37:20.9] JM: Definitely. Interesting. It's an interesting development. What about load balancing. How does load balancing work in Envoy?

[0:37:28.2] MK: Yeah. From the load balancing perspective, we have a couple of different algorithms that we currently support, from basic round robin, to random, to —

[0:37:40.2] JM: Sorry to interrupt, but I guess in this context, the Envoy works in that — Envoy is useful and that all — It can load — All the Envoy instances on each of the hosts of a specific service can talk to each other and get an idea of where to distribute load, and Envoy is this point of communication between each instance of the service is quite useful.

[0:38:04.1] MK: Yeah. Right now, we actually don't talk between Envoys. We don't, I would say, share load information. That's something that I actually think that we will get into probably in the next kind of like three to six months. Right now, we kind of treat it as a very simple, kind of embarrassingly parallel system. Each Envoy kind of makes independent load balancing decisions. That's not to say that it's still not complicated though, because we've got these different algorithms, and one of the things that we do support in the Envoy load balancer, which I think is pretty cool, is that we support what we call zone-aware load balancing.

If you look at a lot of the modern architectures, if you're running in AWS, people typically run their applications in multiple fault tolerance zones, right? Amazon has availability zones, other club providers have kind of similar things.

Not only from a performance perspective, because, typically, it's faster to kind of send request to your local zone. It can also be cheaper. For example, in the Amazon case, they actually charge you when you send traffic between zones. If you're doing kind of very naïve load balancing, you can wind up in the situation where you're balancing your request across kind of all of your service nodes that might be in kind of multiple zones, and that might lead to those requests being both slower, as well as more costly.

Our load balancer in Envoy is actually zone-aware. What it will do is that it will attempt to send traffic to the local zone first and then to kind of achieve good balance, it will overflow traffic properly into other zones. Then, if things start failing, it will properly kind of send traffic amongst all zones. That's something that we use at Lyft that is pretty cool.

[0:39:58.7] JM: It is cool. What about circuit breakers? Maybe you could briefly review the circuit breaker pattern.

[0:40:04.4] MK: Yup. The idea behind circuit breakers is that it's almost always better in a service-oriented architectures to fail as fast as possible, and then to propagate failure back. The reason that we want to do that is that the sooner that you can fail, the sooner that you can apply back pressure. That back pressure might be seven layers deep, like all the way from your database, all the way back to your client, running on some mobile phone. If you don't apply that

back pressure efficiently, you can wind up kind of in this death spiral situation where all of these requests end up basically stacking up, but then the system essentially is not able to recover.

The circuit breaking pattern, just like a circuit breaker in your home electrical system, the idea is that if the load gets too great, you basically fail fast. You essentially attempt to fail as fast as possible and then kind of propagate those failures back.

Envoy supports different types of circuit breakers. We support max concurrent requests, max connection, max concurrent retries. I think those are the ones that we support. I think max pending requests also in the load balancer, but those kind of four primitives allow us to fail very quickly and then basically kind of send those requests back.

We also support an outlier kind of detection pattern, where we will look for upstream hosts that are kind of behaving badly. Right now, what we support is, I would say, fairly simple in the sense that we'll look for a number of consecutive failures. If an upstream host say response with three 500ths in a row, we'll basically take that host our rotation. Then, in the next couple of months, we're going to start supporting similar sophisticated things around outliers in terms of success rate, latency, stuff like that. The main idea around kind of this pattern is that you want to identify failure as fast as possible, and then you want to propagate that failure back.

[0:42:07.4] JM: That's pretty cool. Just naïve question; for different services, why would you have different circuit breaker patterns in different retry policies for how that service would operate if it was flaking out or if it sensed a downstream or upstream problem?

[0:42:24.0] MK: Yeah. In general, we do try to have same defaults that we ship with, because like we were saying before, one of the main things that this type of system provides is that you, as the application developer, you can come and hopefully get most of this stuff for free and actually not have to think about it.

For most of our services, we do actually have defaults, whether they'd be for timeouts, or whether they'd be for kind of circuit breaking max settings. There are cases where you do sometimes have to override them. The cases where they typically have to being overwritten is when services — Is when their request pattern tends to not look like the others. By that, I mean,

let's say that you have a service that, for whatever reason, doesn't use a lot of CPU and requests, for example, take 10 seconds or something like that.

If your circuit breaking patterns are set up to assume that you're handling X simultaneous requests, and those request tend to be CPU bound and those requests typically respond in something like 100 milliseconds, you can kind of do the math and you can figure out what the max concurrent requests are before you start breaking. If your service doesn't fit that pattern, that's when you might have to customize.

What we find typically is that, I would say, for most organizations, 90% or more of services can typically use kind of same defaults, and then there's typically on a few services that are kind of "different" and, for those, we end up customizing them.

[0:44:02.2] JM: I want to get into discussion of observability, 'cause as you've said, the observability is the most important thing that Envoy does. What did you need to build into Envoy to get the monitoring, and the logging, and the tracing, those observability features that you really needed? What did you build to get those right?

[0:44:22.5] MK: Yeah. There are two separate things. First thing is obviously stats. Stats is, of course, a little different from kind of logging and tracing, but stats, meaning, just like counters, engagers, and timers, are obviously super critical from an understanding perspective. It's how most people, for better or worse, kind of interact with these systems.

Envoy emits copious stats, timing and number of active requests, number of total requests, individual response code stats, all of those kinds of things. With those stats that we emit here at Lyft, we actually build kind of templated dashboards that kind of allow us to say, "From any service going to any other service, here's essentially what it looks like." This is kind of one area where the podcast format isn't totally awesome, but I would love to show you kind of a picture.

It's pretty awesome what you can do when you have a system that's outputting identical stats for essentially every Hop, because it allows you to build these really consistent interfaces so that developers, when something goes wrong, they can say, "Oh, I want to look at all the information from the API service to the location service. Let me show you what the timings look like, what

the number of requests are, number of failures, types of failures, all of those kinds of things." That's from the stats perspective.

From the logging and tracing perspective, I would say that the biggest thing that Envoy does is it produces a stable request I.D. that we then propagate amongst the entire system. When kind of a request chain first enters Envoy, Envoy essentially determines if that request already has an existing request I.D. If it doesn't, it basically makes one. If it does, it kind of uses that I.D.

The cool thing about that stable I.D. is that, that stable I.D., we use it both for logging as well as for tracing. From a logging perspective, because we control that I.D., we can do really cool things, where if we do sampled logging, we can make it so that the same request is actually sampled around the entire system. If you sample 1% of requests, say, without doing anything intelligent, it's actually not that useful, because you're going to see 1% of basically random requests on different nodes. Whereas if you sample 1% of kind of requests but you do it in a stable way, you can see 1% of all call chains, which from a logging perspective, can be pretty interesting. That follows throughout client tracing, we're given that request I.D., we can join all of these different trace spans and we can build kind of really nice trace graphs of kind of what's going on.

[0:47:17.1] JM: Yeah. You have also talked about the issues of tail latency, and just for our listeners who want more on tail latency, there's an episode of Software Engineering Radio to a different podcast about — All about tail latency, and this is a really interesting topic. Explain briefly why tail latency is such an important topic, and how you can use Envoy to identify and troubleshoot tail latencies and how that's so useful.

[0:47:43.9] MK: Yeah. I think tail latency, from my perspective, is it's most interesting because it's extremely costly from an operation's perspective. The reason that I say that it's costly is you typically have these SLAs and you're typically defining your kind of SLAs to say that your P-99 latency is going to be within X milliseconds, or kind of something along those lines.

When people are trying to meet that SLA target, they start having questions about, "Well, why am I not meeting it," right? And like, "What is happening to these .1% of requests?" From a networking perspective, or kind of being on a networking team in a company that uses a modern

service-oriented architecture, you wind up spending a surprising amount of time helping people debug and understand where are these requests and why are they failing. That just becomes very costly, and not only is it costly, it can actually deter people from using service-oriented architectures, because people might say, “Well, I don’t actually trust the network,” right? Because the “network” isn’t actually reliable.

From all of the talking that we’ve been doing, hopefully, people will kind of gather that it’s an extremely complicated topic, right? The problem could be in your hardware, it can be in your virtualization layer, it could be in your application, it could be absolutely anywhere.

What Envoy really brings to the table is it brings high performance system that can generate reliable timing stats, and because we generate these stats at every single Hop, you can kind of start to drill in to where your problems are occurring. The thing all of that data basically feeds in to these dashboards that I was talking about, and then it feeds into these tracing systems, and the tracing system that we use here is actually pretty cool. It’s a company called Lightstep, and this company, it actually gives us 100% fidelity.

We actually send each and every request, each and every span to Lightstep. What Lightstep, we can look at P-99.999, right? It can be very useful for people given this data that they didn’t have to do anything to actually generate. It just use Envoy and it’s there, and they can say, “Oh! My service — I failed my SLA target, and I would like to understand where this .1% of request actually — Kind of where they spend their time.”

Given the data that Envoy that generates both from the tracing data, and from the stats, and from the logging, it becomes a lot easier to point people in the right direction. Whereas if they were using either their own libraries, or different systems that had different stats, or different tracing, it would basically be impossible to kind of figure out where this stuff was occurring.

[0:50:50.0] JM: I want to begin a close off by zooming out because I know we’re running up against time. Your background includes work at some companies that have had some really big distributed systems challenges to work on. You worked on EC2 at Amazon, which is like as difficult to distribute system problems that I could imagine. You worked at Twitter, and then now you’re working at Lyft. Can you kind of give a — Maybe not a chronology, but explain how your

perspective on distributed systems, perhaps, observability, or debugging has evolved as your career has progressed and how it feels present day at Lyft.

[0:51:28.7] MK: Yeah, sure. My background is mostly in kind of operating systems, networking, virtualization, like pretty kind of low level system stuff. Then, in the last seven years or so, I've kind of switched more towards kind of the service-oriented architecture networking realm. It's been really interesting to kind of observe how people do kind of deal with these different problems.

When I was at Amazon, obviously, you're kind of dealing with a large company that it's black box, you can kind of invest a lot in kind of some of these components. When I was Twitter, I actually worked on the Twitter front-end proxy, but I worked a lot with the people that were kinda working on Finagle. It was very interesting for me to observe both what was really great about this system and what didn't actually worked that well.

In some sense, Twitter was actually very fortunate, because Twitter was mostly one language, mostly Java, Scala. Twitter kind of had, had or has, the luxury of kind of the single language for the most part, so they can kind of develop this kind of very sophisticated library system.

Then, when I came to Lyft, kind of having seen what was done in the networking realm over at Amazon, and also what was done at Twitter, and then kind of observing that Lyft has three languages that are fully in production, where we're basically migrating off of our PHP monolith, we have people writing services in Python. We have people writing services in Go. We have a couple of C++ services, three or four kind of production languages. It became pretty clear that in a small company, it would just be impossible to write three or four super robust libraries that would do kind of all of the features that we've been talking about over the last hour.

Then, having talked a lot of other companies that are all in similar situations with this polyglot kind of service-oriented architecture, it became clear to me that having this sidecar, this kind of out-of-band proxy could bring huge, huge features, kind of huge, huge operational agility to the table. I think what we've seen here at Lyft is that we've gone from a situation when I joined, coming up on two years ago, where people were literally afraid to essentially make new services. They didn't trust the network. They didn't understand why things were failing, to now,

we are fully gung-ho service mesh kind of service-oriented architecture. I think having Envoy as that substrate layer has given people both the features, as well as the kind of observability to have confidence, that if problems happen, they can actually debug them.

From Lyft perspective, I think it's totally kind of changed the game. Since we've open sourced kind of the recession that we've gotten from other companies, it's been really great. I'm super excited to kind see where this goes, because I think Envoy — It's been awesome for Lyft and I think it can be pretty awesome for other companies also.

[0:54:44.8] JM: Okay, Matt. Thanks for a really interesting conversation. I loved talking about Envoy, there's a lot to learn about both preparing for the show and talking to you. I really appreciate you coming on Software Engineering Daily.

[0:54:56.4] MK: Thanks for having me. That was great.

[END OF INTERVIEW]

[0:55:02.0] JM: Listeners have asked how they can support Software Engineering Daily. Please write us a review on iTunes, or share your favorite episodes on Twitter and Facebook. Follow us on Twitter @software_daily, or on our Facebook group, called Software Engineering Daily.

Please join our Slack channel and subscribe to our newsletter at softwareengineeringdaily.com, and you can always e-mail me, jeff@softwareengineeringdaily.com if you're a listener and you want to give your feedback, or ideas for shows, or your criticism. I'd love to hear from you.

Of course, if you're interested in sponsoring Software Engineering Daily, please reach out. You can e-mail me at jeff@softwareengineeringdaily.com. Thanks again for listening to this show.

[END]

