EPISODE 361

[INTRODUCTION]

[0:00:00.7] JM: Modern software applications are often built out of loosely coupled microservices. These services can be written in different languages by different people, but communication between the services needs to be standardized. For this reason, a service proxy is useful. A service proxy is a sidecar container that sits next to a service and facilitates communications with other services. Once every service has a sidecar proxy, that sidecar can be used as a way to communicate with a centralized control plan.

The sidecar can report telemetry data to the control plane and the control plane can be used to set policies across services, such as rules for scaling and load balancing which might vary from service to service. Istiois an open platform to connect, manage and secure microservices. Istio a service mesh the uses on-voice service proxies. If all these sounds confusing, don't worry, we'll explain it all in today's interview Varun Talwar and Louis Ryan, who both work on Istio at Google.

If you're looking for old episodes of Software Engineering Daily and you don't know how to find the ones that are interesting to you, you can check out our new topic feeds which are in iTunes or wherever you find your podcasts. We've sorted all 500 of our old episodes into categories like business and blockchain and cloud engineering, JavaScript, machine learning, we've got a feed with the greatest hits of Software Engineering Daily. If you have any trouble parsing the feed for the episodes worth listening to, greatest hits is a good place to start.

Whatever specific area of software you're curious about, we have a feed for you. You can check the show notes for more details about these new topic feeds and I'd love to hear your feedback on them. I hope you like this episode.

[SPONSOR MESSAGE]

[0:02:02.7] JM: Your application sits on layers of dynamic infrastructure and supporting services. Datadog brings you visibility into every part of your infrastructure, plus, APM for

© 2017 Software Engineering Daily

monitoring your application's performance. Dashboarding, collaboration tools, and alerts let you develop your own workflow for observability and incident response. Datadog integrates seamlessly with all of your apps and systems; from Slack, to Amazon web services, so you can get visibility in minutes.

Go to softwareengineeringdaily.com/datadog to get started with Datadog and get a free t-shirt. With observability, distributed tracing, and customizable visualizations, Datadog is loved and trusted by thousands of enterprises including Salesforce, PagerDuty, and Zendesk. If you haven't tried Datadog at your company or on your side project, go to softwareengineeringdaily.com/datadog to support Software Engineering Daily and get a free t-shirt.

Our deepest thanks to Datadog for being a new sponsor of Software Engineering Daily, it is only with the help of sponsors like you that this show is successful. Thanks again.

[INTERVIEW]

[0:03:26.1] JM: Varun Talwar and Louis Ryan work at Google. They're currently working on the Istio service mesh project which we're going to discuss in this episode. Guys, welcome to Software Engineering Daily.

[0:03:36.8] LR: Hey, thanks Jeff.

[0:03:38.0] VT: Thanks, Jeff.

[0:03:39.1] JM: Today we're going to talk about the service mesh, and this is something that comes up related to Docker and Kubernetes. First, we should motivate a little bit about kind of the issues that develop when a company moves their application from a monolith to microservices. Maybe they're running their microservices on Kubernetes, and there are these communication issues that develop when you have these different services communicating with each other, and I guess not just communication issues but problems of the holistic application that's running across Kubernetes where you want to have some centralized functionality across those services in addition to having the individual services being decoupled from one another.

Maybe you could talk about some of those issues in communications and the overall health of an application that are not taken care of just by virtue of the fact that you have moved your application to microservices on Kubernetes.

[0:04:50.6] LR: Right, the most obvious thing when you turn a monolithic application into a distributed system, your network is going to break. People like to refer to it as the fallacy of permanent network availability. It's a real thing. It happens an awful a lot. It happens in surprising ways.

The mechanisms that you were using before in your monolithic application are harder to make occur again when you have this distribution. You're used to using consolidated logging, or you're used to using section handling and writing retry loops that way. When the functionality is spread across all these disparate processes, the scope of failure modes increases pretty dramatically. Not just a call simply fail, you also have to deal with the fact that calls may partially succeed, which is almost a harder thing to deal with.

What's happening as you're breaking the monolith apart, what's also often happening is you're also becoming more polyglot, and so the programming practices that you used to have in one monolith don't apply anymore. People are using different programming practices. You've broken responsibility across many teams, and so you don't have this uniform toolset to start working or corralling these behaviors again, and so it become extremely expensive to kind of recreate some of the standard management practices that you used to have in the monolith in this distributed system.

What service mesh is attempting to do is to say, "Look, there are these application level concerns around traffic reliability around observability that you've essentially moved over into the networking layer but you haven't figured out how to standardize work for them yet." Service mesh is trying to do that for you.

It's trying to give you, effectively, a whole bunch of behaviors that appear like they're part of the network and it doesn't really matter how they're implemented, but that you can rely on them to improve the reliability and behavior of your distributed system.

[0:07:00.5] JM: I think I can break down what you just said into two broad categories, which is that you want some increased reliability/observability/richness of features, distributed logging, for example, aggregating logs instead of getting them all from one place, a monolity. Then you also want this standardization because you have a polyglot system. How are we going to to standardize how these different things are communicating with each other?

I think the latter of those two problems we addressed in a show that I did with Matt Klein from Lyft. He built Envoy, which is a service proxy that is heavily used within the Istio service mesh that we're going to discuss.

Explain what a service proxy is.

[0:07:47.0] LR: A service proxy is this piece of functionality that is attached to your application. WE attach it to the application because, effectively, it's acting on behalf of the application. People often use the term sidecar proxy, because this proxies is effectively deployed along with your application logically. It's like a tool that the applications is using. Ideally, maybe the application doesn't even know that it's using this tool, but it's a bridge between the application on the physical network that adds in all these additional capabilities.

One of the reasons why we deploy it as a sidecar proxy rather than maybe as the traditional middle proxy architecture that people might be used to, because this proxy is acting on behalf of the application, one of the features that it's going provide is it's going to act using the identity of the application, and that often means credential distribution. It means that it has access to secrets that represents the identity of the workload on behalf of whom it's going to be performing certain actions. rom a security perspective, you want to make sure that you don't have confused deputy problems. It's very much easier using the sidecar model to make sure that doesn't occur.

Now that you have this sidecar proxy, it's sitting next to your application. It's part of the application logically even though it may actually be a separate process. All the application traffic is flowing through it. It can start to do a lot of these functions that I just talked about that you mentioned; improving reliability, increasing observability.

[0:09:18.8] VT: To add to that — Just to explain, I guess, new users, this model of a co-process or a process sitting next to the application and basically acting on behalf, as Louis said, is what makes this possible and independent of language. That's what gets to the polyglot to really reflect no matter what your application is written in. This process can do a bunch of reliability of observability things on your behalf. I think the other one being, since it's sitting next to the application, can use the application identity on all sort of outbound traffic from their application.

You do that in — Where you get uniformity and consistency is when every service in the application, the overall system is using the same way. That's when those combined together is what we call the service mesh.

[0:10:22.3] JM: Right. To give a finer point, to reiterate what you both just said about the service proxy, it sits in a sidecar which is a term used to describe, in this case, a container that sits alongside another container in a pod. In Kubernetes, you have pods, and a pod is like an abstract —Am I phrasing that correctly?

[0:10:43.5] LR: Yeah. In the context of Kubernetes, that's typically how we would run. There are obviously other deployment vehicles. You may just have your service running on VMs, in which case the service proxy would run as a sidecar, or i.e; another process, next to whatever other processes are running inside that VM.

[0:11:03.8] JM: Right. The main thing to focus on, I guess, is the locality of this sidecar because the sidecar is close to the service because the service — whenever the service — In a microservices architecture, you're going to want to communicate between services a lot. You have these service calls that go from service to service to service in order to fully fulfill the user's request. When you have these different services communicating with each other, it's useful to have this standardization.

Every time a service is going to communicate with another service, you have it speak through the proxy and one proxied service point will talk to another proxy rather than the service is talking directly to each other. Because every service has basically this microphone that it's talking through to other services, you can have those microphones talk to each other. Basically,

that's the service mesh and all those different sidecars can report back information to a centralized place.

Explain what that, I think, centralization or that collection of information. That's what defines the service mesh. Talk more about what we have architecturally in addition to these sidecar service proxies that we would talk about as the service mesh.

[0:12:38.5] LR: The proxies provide what we would typically term the data plane. They sit between the services, the traffic flows through the proxies as it moves between service to service, and the proxies do two basic functions. They make sure that the traffic flows well and reliably, and then they also extract features out of that traffic that then you could use to start enforcing policy on or gaining insight into the behavior of the mesh by adding observability too. What the proxies do in Istio today is they make calls to a centralized service that's effectively taking these signals and emitting policy decisions back into the data plane.

One of the reasons you want centralization is particularly of policy, is that as an operator, when you're managing a service mesh, there's a variety of things that you want to achieve in a common way no matter what service is talking to what other service. Maybe you have an investment in an existing APM vendor or logs provider and you want to make sure that every service, all their logs are going into the right location. The mesh can help facilitate that because it gives you this centralized point of control where you could send logs or telemetry data.

Perhaps more interesting in the long run for people though is as your deployments become more complicated, you want to enforce policies around which services are allowed to talk to each other's services. What types of traffic are allowed to flow between those services? Not just services maybe between physical locations, maybe you have some compliance requirements. As long as you can guarantee that those policies that you want to enforce are being respected by the services, then you have a high confidence that you're meeting your organizational goals.

The mesh enables you to do that type of thing because it has given you this uniform way to interpose into traffic. You can reliably know that your traffic is conforming to those organizational policies. Usually those organizational policies are managed centrally. We have a component in Istio called a mixer which is where really the kind of the mesh or cluster operator goes and

defines those policies, plugs in integrations into downstream systems, whether it's a logs collection system, the telemetry system, or maybe it's an ACL system, like maybe you want to write an integration into active director or something like that. You want to manage it essentially because these are cross-cutting concerns. They are things that operators or SRE roles in your company are responsible for. Maybe you have to do audit or compliance recording for a CIO. You want to make sure that you're covering these things, and you don't want to have to do that for every process or every keys of application that's running in your data center or on premise or in cloud, and so this gives you a way to start to rollout those types of things.

[0:15:27.5] JM: Examples of problems that you might solve by policy, would these be things like security, and load-balancing, and monitoring, and rate limiting?

[0:15:39.9] LR: We try to be careful how we use the term policy, because it can be — It's an overly broad term unless you're a little careful how you use it. We tend to think of policy as things like quota and ACL. Organizational policy, there are things like compliance requirements. It's hard to say that auditing is a policy per se in the kind of runtime sense, but it's a policy that your organization might have that you must audit.

We try to separate runtime policy things that affect the behavior or services when they talk to each other, and policies that your organization is trying to achieve or ensure compliance with. It's slightly different things, but when we use an umbrella term, we do tend to use the term policy to describe those type of things.

Telemetry is a little bit different. Telemetry is hugely — Its own consideration people tend not to refer to telemetry as policy. We tend to use slightly different terminology when we talk about that. Effectively, from an organization policy perspective, you want to make sure that you're doing all of these things. That's what this really helps you achieve. It helps you achieve consistency of all of those cross-cutting concerns across the entire fleet without having to go and modify every single application in the fleet to make that happen.

[0:16:57.6] VT: To add to both your previous questions, sort of why centralization and what problems that that solves, I think one of the things that is happening with this, "Oh, I want to break monolith into microservices that create agility for each dev team," what we're trying to do

with service mesh is saying, “Hey, what about the dev ops team and the SRE team?” That’s usually not given — Associated with every team that’s typically managing the overall system across the organization.

In terms of — I think there’s two answers to why centralization, like one uniform toolset so to speak for the dev ops and SRE folks, and that is the injection point for either runtime policy or [inaudible 0:17:46.9] policy. The second piece is one point of integration for telemetry.

Now, even though you have end teams building in end languages and services and you want to give them that freedom of evolving those at their own pace and building in with the architecture they want, you still want to have a unified view of end-to-end latency and matrix and you wouldn’t want to have end integration into monitoring systems run by each of those end teams.

Basically, the other piece that centralization gives you is one point of integration, and that helps from standardizing the nouns that the entire team is seeing around the metrics as well as the call chain problem of service to service to service and then debugging those.

[SPONSOR MESSAGE]

[0:18:47.7] JM: Angular, React, View, Knockout, the forecast calls for a flurry of frameworks making it hard to decide which to use, or maybe you already have a preferred JavaScript framework, but you want to try out a new one. Wijmo and GrapeCity bring you the How to Choose the Best JavaScript Framework For Your Team e-book.

In this free e-book, you'll learn about JavaScript frameworks. You'll learn about software design patterns and you’ll learn about the advantage of using frameworks with UI libraries, like Wijmo. You’ll also learn about the basic history and the purposes of JavaScript’s top frameworks. You’ll also learn how to integrate a WIjmo UI control in pure JavaScript and in some of the top JavaScript frameworks that we’ve already were talked about, like Angular, React, View and Knockout.

Wijmo’s spec method allows you to determine which framework is best suited to your project based on your priorities. Whatever those priorities and your selections are, you can learn how to

migrate to a new framework in this e-book. Best of all, this e-book is free. You can download your copy today to help choose a framework for your work at softwareengineeringdaily.com/grapecity.

Thanks to GrapeCity for being a new sponsor Software Engineering Daily, and you can check out that e-book at softwareengineeringdaily.com/grapecity.

[INTERVIEW CONTINUED]

[0:20:31.6] JM: Distributed tracing, for example seems like something where this is a relevant conversation, because just distributor tracing, you want over all of your services. I can't think of a reason why you would want a service not to have distributed tracing. How would that fit into this conversation? Is there some well-formed integration that you would do, I think, just as an example of something that you would use Istio for or use a service mesh for? Does distributor tracing fit into that example?

[0:21:08.9] LR: Yeah, I think it does. Distributing tracing, it's a complex problem for people to solve well. There's a couple of different issues that you need to address. I broadly categorize those as kind of instrumentation and then triggering.

On the triggering side, what you really want is to be able to control when and how traces become initiated. It can be very expensive to have tracing on all the time. It can produce far more data than you would be willing to actually store and it can severely impact how much data is flowing over your network if you're gathering very very detailed traces for every small operation that occurred. It might not be a good idea, for instance, to enable distributed tracing for everything that was going to make a thousand calls to Redis, that would probably be the bad idea.

Triggering is important. You want to be able to control when it occurs, how often it occurs, and that's something that Istio can definitely help with, because if we control the ingress points into the mesh, the traffic between nodes in the mesh. We can have operators define triggering policies that apply across the fleet. We can standardize the mechanisms by which triggering occurs so that you don't have to go and write custom-triggering code in different applications.

That's one area that we can help with significantly. That is actually a very very important concern.

The second part instrumentation we can help a little bit. Ultimately, some of the things that you're going to want to trace occur inside applications themselves, and so there are libraries that exist in runtimes that help with that problem, so it can be a pretty good example of that, but there are others. There are standardization efforts out there, like open tracing and others that we are supportive of.

What Istio can do — We can't actually modify or enable tracing of behaviors with inside the application. We can at least make sure that as those traces propagate down to the system that we enrich them with context that we have in the mesh, that you wouldn't have to go and alter your application code for. We can make sure that we're accurately and reliably layering in information that we have about the communication that occurs between services to make sure that those traces have a good level of detail in them. We can do that augmentation both as the traffic flows through the system, but also at trace capture time.

Ultimately, in distributed tracing systems, nodes within the graph are reporting trace spans into some centralized system for later storage. One of the things we can do is, basically, capture that event and do enrichment there even for traces where we weren't in the network path, because we know a lot of the topology of a network, we can make sure that we are pushing some of that information into traces that were captured by applications themselves.

Those are the kind of two basic points of value add. Of those two I would say uniformity around triggering is probably the most important thing. The last thing you want to have is you rollout an application to production and then realize that you need to enable or trigger a trace after the fact, but you have no way of doing it. That would be a pretty terrible outcome.

[0:24:16.1] JM: I want to spend a little more time talking about the deployment and topology of Istio. If I've got my Kubernetes cluster, I've got a bunch of containers running in pods somewhere — We've kind of gone over in detail the idea of a sidecar, these envoy sidecar service proxies that are talking to each other talk. Talk a little bit more about the control plane, which is this centralized entity that can do lots of things, it could synchronize different things,

can aggregate different things. Talk about the relationship between that data plane and the different sidecar proxies that are aggregating this information.

[0:25:07.8] LR: You deploy the Istio, you've created a mesh, you know how all these application is running. What you really want to be able to do as an operator is to be able to affect changes in how traffic flows through the network without requiring people change those applications. That's maybe the thing people think about first when they think about service meshes. You need some way of managing the configuration of that, so we've provided a system to do that.

[0:25:34.8] JM: Sorry to interrupt you, but you mean without doing a deployment.

[0:25:36.8] LR: Right. Without having to redeploy or update application code.

[0:25:41.2] JM: You want to change an application, but you don't want to do a deployment. You can use Istio.

[0:25:47.1] LR: Right. A great example would be something like A-B testing or migrating traffic from a legacy service to a newer version of that service or just some other vendor. These are the things that you want to be able to experiment with without really having to radically change your deployment architecture, and Istio gives you the kind of fine grain traffic controls to say, "Look, I want to siphon off 1% of my incoming traffic and send it to this alternate backend to qualify it for release management." We're talking about canary deployment scenarios. That's a very — Kind of maybe the canonical example for service mesh traffic management.

Other examples might be you want to change how you do load balancing on the fly because you're having service availability problems. Maybe you do need to stand up another cluster or some more instances or do something else with regions. All your incoming traffic or all services going between traffic is currently going to one location and you need to make sure it goes elsewhere because you're having stability problems.

Service mesh would help you do those types of things, again, without having to change all the clients in the network. There's a lot of scenarios that we've had experience with at Google where you have traffic and balancing and you need to be able to react to it quickly. Rubbing out

application changes is the exact opposite of doing something quickly from an operations perspective.

[0:27:09.8] JM: Is the data plane pulling the different sidecar? What's the strategy for aggregating information?

[0:27:20.3] LR: When you say aggregation, typically — There's s pushing out changes down into the service mesh which effectively means making a configuration change to how traffic is supposed to flow and then distributing that to all the proxies in the mesh that need to be aware of that change. That is effectively done by the proxies pulling for that information.

[0:27:40.6] JM: Oh, is see. The proxies are asking the control plane, "Hey, what's the policy right now?"

[0:27:47.2] LR: Right. Yes. We try to do that as efficiently as possible, because the mesh could be very large. You would have tens of thousands of proxies asking those types of questions. You need to be able to scale up to that. Similarly, the proxies are also providing feedback to the mesh. As traffic flows to the proxies, it's observing whether API calls are failing, is latency spiking? Those types of things. By collecting the telemetry and feeding that back into the system, you can drive things like horizontal auto-scaling. We haven't done this yet with Kubernetes, we absolutely plan to provide standardized metrics about pod scaling latency requirements so that you could horizontally auto-scale pods based on latency instead of just CPU consumption, which is pretty important for services that have specific types of SLAs. That's information flowing in the other direction. Basically, observing the behavior of the data plane and using it to control either routing and or scaling of deployments themselves.

[0:28:47.0] VT: Typically, to make it even simpler, all of the service proxies envoy's in Istio have pretty — They're obviously talking to the control plane in both directions, as Louis said, from proxy to control plane is pretty simple and standardized API. They're asking for — Typically, you're either asking for a check or a report. You're either saying, "Should I allow this traffic or not based on all the policies?" Or you're saying, "I want to report all the attributes and what's going on." That's the proxy to control plane.

The other way which once you are observing everything, you can start doing. We haven't done those, but those are more interesting things we can get into in future releases.

[0:29:44.9] JM: I've done a couple shows about Prometheus, which is this distributed monitoring tool. It's based on Google's Borg mon. Kubernetes was based on Borg. Prometheus was based on Borg mon. Some of the techniques that we're talking about here sound somewhat similar to the problems that Prometheus is trying to solve. Can you talk about the comparison between Prometheus and a service mesh?

[0:30:17.3] VT: Prometheus is primarily just, at least the way we look at it, is one of the possible sort of monitoring visualization tool. Prometheus, I would put under the bucket of sort of monitoring and visualization and everything upstream on top of alerts, etc., on top of that.

The way mesh is more runtime in terms of what gets injected in with your application to actually — And be in the runtime to collect signals that you can decide to put into Prometheus or some other similar system, their idea of those and commercial space. That's a distinction I see.

[0:31:07.5] JM: Can Istio do the job, do some of the work — Or maybe the Envoy sidecar, depending on what part of Istio you want to talk about. Can it do some of the work of handing off information to Prometheus? Maybe you can talk about if there's any synergies there.

[0:31:27.3] VT: That's exactly what we are sort of doing. What Istio is doing is —

[0:31:32.0] JM: It seems quite useful.

[0:31:33.6] VT: Yeah, our current release is basically signals from Istio flowing into the Prometheus tool where you can start to see graphs around QPS and error raid and latency from service to service. Really, sort of application L-7 level metrics that you would like to see when you're holistically over and above all of the L-3, L-4 CPU resource utilization metrics that you already see.

The way I see it, we're enhancing the kinds of metrics that you can see in Prometheus or any other similar system. It's exactly one of the things that people want when you're running large —

When you start splitting and having it into large microservices is really getting the confidence that my latencies by breaking things apart is not killing and having those signals given back to you.

The other thing I would add, which is when you do traffic management things, especially like A-B testing, stage rollouts, you also want instant as quickly as possible feedback. If I just rollout N % traffic from A to B to a newer version of B, you want to get the confidence about the latency for that N% traffic is under control. As you do these rollouts, that's where I think the policy piece ties in with the metrics space and there's more interesting things we can do there.

[0:33:17.2] LR: Yeah. This kind of brings back to something that I wanted to touch on. One thing that Istio offers about anything else, or service or interest in general, is they offer some uniformity across the entire fleet for these metrics. As we mentioned, Kubernetes and other things tend to — They give you uniformity for network bytes in and out, CPU around consumption. When it came to metrics about L-7 behaviors, you're kind of at the women mercy of how well a job any given application in fleet did integrating telemetry.

If you have gross inconsistencies in telemetry across the fleet of applications that you're reserving, it's very hard to reason about behavior. It's very hard to plan change. It's very hard to manage rollouts, to do all these types of things. Having the ability to enforce some uniformity is really what allows the operator to make decisions. If I had to say through my experiencing working at Google, the thing that most struck me when I was managing production systems was how uniform metrics were about service behaviors and how important it was to be able to get that information to make decisions.

[SPONSOR MESSAGE]

[0:34:35.7] JM: Artificial intelligence is dramatically evolving the way that our world works, and to make AI easier and faster, we need new kinds of hardware and software, which is why Intel acquired Nervana Systems and its platform for deep learning.

Intel Nervana is hiring engineers to help develop a full stack for AI from chip design to software frameworks. Go to softwareengineeringdaily.com/intel to apply for an opening on the team. To

learn more about the company, check out the interviews that I've conducted with its engineers. Those are also available at softwareengineeringdaily.com/intel. Come build the future with Intel Nervana. Go to softwareengineeringdaily.com/intel to apply now.

[INTERVIEW CONTINUED]

[0:35:32.1] JM: I worked at Amazon for a while and I had a similar — I tried and operate things at that level of sophistication or authority that you did or that you have, but I was kind of blown away and impressed by the fact that there was standardization. They were able to impose standardization by offering — It was something similar to a sidecar, and this was before there's a lot of open-source attention around this kind of infrastructure. It's fun, because when you work on one of these companies, you really do get to see the future of infrastructure before it gets released to the open source world, because I was seeing it and I was like, "Wow! What is this? This is mind-blowing. This is magical." Now, I'm reporting on it, I kind of understand, "Oh, okay. It's this thing. It's very impressive, but I kind of — It's a little demystifying."

[0:36:24.7] VT: Right. The goal is obviously to bring a little of that capability to the community in an accessible and easy to consume way. One of our goals in Istio is to be able to have people turn on Istio without having to do anything to their application, without having to rewrite it, without having to do any of those things, and to get this information out.

Once we can get that information out easily, then you can start to appreciate the value that it brings and you can start to rely on it and make operational decisions based on it. I'm sure that was your experience with Amazon, like if you were trying to decide, "Do we need to scale this service up? How many more instances do we need to bring up to meet our SLO?" All those types of things. This really really helps with that problem.

[0:37:06.8] JM: It does. I would say one reason is helps is because it reduces the friction to doing X, like if X is add some servers or do some better load-balancing or do a canary deployment or something, if part of that decision is how long is it going to take me to actually do that or how many hoops am I going to have to jump through to figure out how to do that, it's a little more annoying and it creates some bias towards not doing something that you should do.

There really is a lot of importance with this ease-of-use, and I know that I think part of both your goals is to get Istio well-integrated with the platform as a service offering like the Google container platform as a service offer and the Kubernetes offerings, I guess, of different platform as a service flavors, getting it to that kind of a one click glory easier to use.

Actually, I want to talk about hour talk about kind of a higher level both the open source community and close source platform as a service stuff, but I guess before we kind of shift the conversation to talking about this broader ecosystem, if there's somebody listening right now and they are a developer, if they're got a Kubernetes cluster, they work at a company that has a Kubernetes cluster, I guess just put capstone on the conversation that we've had so far.

What are the technical problems that this type of developer might be having where they should look into Istio, they should consider it even at the sort of nascent — It's not nascent, but this kind of 1.0 release, I don't know, if you have 1.0, but the state of Istio today. What kind of person should be looking into Istio?

[0:38:48.7] LR: I would say everybody. No, I don't mean to be facetious there. You could almost think of Istio, like in the context particularly of Kubernetes as just a bunch of networking tools that will help you make your applications, your distributed applications more reliable to build. That's probably your first port of call. You've done some stuff that Kubernetes which probably means you're doing something micro-service-ish. We actually don't care how micro your services are. You're trying to figure out, "Am I going to go and put a whole bunch of retry logic into my code? I know the application fails, or the networking is going to failure modes and I need to deal with those." Maybe you're not too concerned with that problem, but you are looking around, "How am I going to do AMP? How am I going to track the behavior of my applications in the network? Am I going to have to start writing a bunch of instrumentation code?"

You might find it a lot easier and cheaper just to go and turn Istio on and rely on its kind of automated instrumentation mechanisms to get a pretty holistic sense of what's going on in your network with a very low investment. If you're out in this kind of nascent stage, that that's what should be motivating you, is that you don't have to make a large investment to get a very significant amount of value back out.

Once you've done that and I've gotten used to the concepts and have gotten used to the extra value that that stuff is going to bring, then you might want to start looking at the additional value that we think applies to deployments as they mature where you want to start having some more policy control. Your first protocol should be, "Hey, I need observability. Observability can be hard. Here's how I get a whole bunch of it," and in a really consistent and very well-designed way for free effectively.

[0:40:36.0] JM: I hear the bell tolling with our calendar invites for 10 minutes away, so I'll be crisp with your time for the remaining questions. One thing I find interesting and sometimes confusing is the boundaries between what problems Kubernetes should be solving and what problems the open source, like the look the other open-source projects, perhaps the Cloud Native Computing Foundation projects should be solving, how do you guys look at those boundaries? Why shouldn't Istio, for example, be in the purview of Kubernetes? If everybody in Kubernetes should be looking at these issues and perhaps considering bundling them into their Kubernetes cluster, why shouldn't this problem be solved within the Kubernetes project itself?

[0:41:23.7] VT: I think there's a simple answer there, which is all of the concerns mentioned that Istio addresses mentioned in our conversation pretty much exist maybe in slightly different flavors even if you're sort of running in VMs or bare metals or other orchestration environments. The problems around making — Inherently making applications more reliable without doing all of the heavy lifting in code about consistent observability, security, everything that we mentioned, they're applicable everywhere.

That's sort of the reason of why not just in there, although our first target environment has been Kubernetes and depending on community response, we may see more and more of Kubernetes plus Istio deployments than just Kubernetes deployment.

As Istio project, we would like to give the capabilities of mesh to wherever you're running. Of course, we would like more and more people to be running on Kubernetes. In fact, I think mesh can even help in that journey of you moving to Kubernetes and we would like to help there, but it's going to be a process and people are — There's lots of workloads running outside Kubernetes as well.

[0:42:46.2] JM: I did a show about Linkerd recently, which is another service mesh that's built in this space, and this raises the question — Another boundaries question. In this open source ecosystem, there are different players that have some overlapping functionality, and sometimes that overlap could be complementary. For example, I think the Prometheus and Istio overlap, it sounds like Istio can solve some of the problems for Prometheus, and that's a way that's harmonious. There are other places where there seems to be some conflict. How do you guys assess that diplomatically when there are projects with overlapping functionality?

[0:43:32.2] LR: I think there's a variety of things. Linkerd and Istio, they overlap in some areas and they don't overlap in others. In the areas that we don't overlap, we can actually be complementary to each other, particularly with regard to some of the policy stuff that we talked about earlier. There's definitely a potential for complimentary development there.

In the kind of nitty-gritty details of networking and data path, we are directly overlapping, maybe not competing, because maybe people view these products slightly differently or place different requirements on them, but it's certainly seem the same question.

Envoy, which is the sidecar proxy that we use, and Linkerd which is the sidecar proxy that the Buoyant guys developed and there are other folks out there using jproxy or Nginx or homegrown stuff to do these types of things, we view that as one market validation, but also having a healthy ecosystem and a healthy competition will probably help us thresh-out requirements and make progress more quickly.

From my view ultimately, proxies should become a commodity technology that as long as a proxy is providing you the features that you want in a performant scalable and reliable way, then if you move between environments and you've happen to get a different proxy when you moved to another environment, that things continue to function, and that there are diversions. That's what's most important.

If you were running your workload on Kubernetes on premise and then you move it over to Google cloud platform or IBM Bluemix or something like that, that things keep working even though the cloud vendor is now managing that behavior for you. I think that's what people really

want, is they don't want to deal with divergence and they don't want to pay too much for it as well.

We chose Envoy because we thought it represented a good mix of features and performance and reliability in community, which is why we've made a strategic bet on him, but we don't see everybody agreeing with that in all scenarios. What we do want to try and do is work with these other folks and come to some agreement around common behaviors and things like that, because that's ultimately what our customers want.

[0:45:45.1] JM: All right. Guys, I want to thank for coming on Software Engineering Daily. I know we had some struggles getting the show organized and orchestrated, and it's the middle of the afternoon, you guys are both in the midst of busy days. If you have something in the future, if you guys want to come back on, I'd be happy to have you. This is a really interesting subject and I know we barely scratched the surface, and this of this whole area is moving so quickly, so I'm sure we'll have something to talk about in a month, if not a week.

[0:46:13.7] LR: Yeah, next week. I'd certainly be more than happy to come back on. Yeah, this base is moving very quickly. We're busily trying to keep up with it ourselves, and so we know that it might be a struggle for the community out there to kind of keep tabs on what the major trends are and what they should be paying attention to. We want to stay in touch with people and I think this is a great vehicle for that.

[0:46:37.3] JM: It is multicast.

[0:46:38.5] VT: Thanks for having us. It gives us a channel to let people know what's going on. Thank you.

[0:46:43.7] JM: Absolutely. Okay, well thanks guys.

[END OF EPISODE]

[0:46:51.7] JM: You have a full-time engineering job. You work on back-end systems of front-end web development, but the device that you interact with the most is your smartphone and

you want to know how to program it. You could wade through online resources and create your own curriculum from the tutorials and the code snippets that you find online, but there is a more efficient option than teaching yourself.

If you want to learn mobile development from great instructors for free, check out CodePath. CodePath is an 8-week iOS and android development class for professional engineers who are looking to build a new skill. CodePath has free evening classes for dedicated experienced engineers and designers. I could personally vouch for the effectiveness of the CodePath program because I just hired someone full-time from CodePath to work on my company Adforprize. He was a talented engineer before he joined CodePath, but the free classes that CodePath offered him allowed him to develop a new skill, which was mobile development.

With that in mind, if you're looking for talented mobile developers for your company, CodePath is also something you should check out. Whether you're an engineer who's looking to retrain as a mobile developer or if you're looking to hire mobile engineers, go to codepath.com to learn more. You can also listen to my interview with Nathan Esquenazi of CodePath to learn more, and thanks to the team at CodePath for sponsoring Software Engineering Daily and for providing a platform that is useful to the software community.

[END]

© 2017 Software Engineering Daily