EPISODE 492

[INTRODUCTION]

[0:00:00.3] JM: A single user request hits Google servers. A user is looking for search results. Maybe they're looking for cat pictures. In order to deliver those search results, that request will have to hit several different internal services on the way to delivering a user response.

These different services work together to satisfy that user request. All of those services need to communicate efficiently. They need to scale and they need to be secure. Services need to have a consistent way of being observable, allowing for logging and monitoring among those different services. Services need to have a proper management system for security.

Since every service wants these different features, like communication, load balancing and security, it makes sense to build these features into a common system that can be deployed to every server.

Louis Ryan has spent his years at Google working on service infrastructure. During that time he's seen massive changes in the way that traffic flows through Google. First, there was the rise of Android and all of the user traffic from mobile phones. Second, there was the rise of Google cloud platform, which meant that Google was now responsible for nodes deployed by users outside of Google.

These two major changes mobile and cloud lead to an increase in the amount of traffic and the type of traffic. All of these traffic leads to more internal services communicating with each other. How to service networking change in such an environment, and that's one thing we're going to discuss today.

Google's adaptation to these new networking commissions was to introduce a service mesh. A service mesh is a network for services. It provides observability, resiliency, traffic control and other features to every service that plugs into it. Each service needs to plug in to the service mesh. In Kubernetes, services connect to the mesh through a sidecar.

Now let me explain the term sidecar, because this is pretty important for people who are learning about Kubernetes. Kubernetes manages its resources in pods. Each pod contains a set of containers.

You might have a pod that's dedicated to responding to any user that is requesting a picture of a cat. This is the pod abstraction is dedicated to this. Within that pod, you not only have the container that is going to help you serve that cat picture. You don't only have this application container. You also have other sidecar containers that help out that application container.

You can have a sidecar that gets deployed next to your application container that handles logging, or a sidecar that helps out with monitoring, or network communications. If you're using the Istio service mesh, which is what Louis works on at Google, then that means you're using a sidecar called envoy.

Envoy is a sidecar that's called a service proxy, and it provides configuration updates, load balancing, proxying and lots of other benefits. We've done some shows about envoy, if you want to check those out. If we get all of those features out of envoy, like we get a load balance or proxying, we get all these benefits out of the sidecar proxy container, why do we need these other abstraction, this service mesh?

Well, that's because it's helpful to have a tool that aggregates and centralizes all of the different communications among these different proxies that are deployed as sidecar containers. Every service gets a sidecar for a service proxy, and every service proxy communicates with the centralized service mesh.

Louis Ryan joins this episode to explain the motivations for building the Istio service mesh and the problems it solves for Kubernetes developers. The next two weeks are devoted exclusively to the world of Kubernetes and the surrounding projects.

If you are looking to learn more about these projects and you want to get some exposure to our past episodes, you can check out our apps for iOS or Android, where we have all of our episodes, all 650 episodes. The podcast players that you'll find in the app store are only going

to give you the most recent 100 episodes, but these apps have all of our episodes, including our old episodes about Kubernetes, if you're looking to get an introduction to it.
With that, let's get on with this episode of Software Engineering Daily.

[SPONSOR MESSAGE]

[0:04:43.2] JM: As your container simplifies the deployment, management and operations of Kubernetes. Eliminate the complicated planning and deployment of fully orchestrated containerized applications with Kubernetes.

You can quickly provision clusters to be up and running in no time, while simplifying your monitoring and cluster management through auto upgrades and a built-in operations console. Avoid being locked-in to any one vendor or resource. You can continue to work with the tools that you already know, so just helm and move applications to any Kubernetes deployment.

Integrate with your choice of container registry, including Azure container registry. Also, quickly and efficiently scale to maximize your resource utilization without having to take your applications offline. Isolate your application from infrastructure failures and transparently scale the underlying infrastructure to meet growing demand, all while increasing the security, reliability and availability of critical business workloads with Azure.

Check out the Azure Container Service at aka.ms/acs. That's aka.ms/acs. The link is in the show notes. Thank you to Azure Container Service for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:06:10.5] JM: Louis Ryan is one of the lead developers on Istio. Louis, welcome back to Software Engineering Daily.

[0:06:16.5] LR: Hey, thanks for having me.

[0:06:17.8] JM: You are a principal engineer at Google and I saw you give a talk at QCon about some of your historical experiences at Google and how those experiences impacted and led to you starting the Istio project. For many years, you've been working on API infrastructure at Google. What this means is HTTP traffic comes in and this is the layer at which traffic is routed and distributed to different services, it's load-balanced. Give a description for the architecture at Google that you have spent your years on. What goes at the API infrastructure layer?

[0:07:01.8] LR: Right. I mean, like most API infrastructure products, there's some proxy in the path between the client and ultimately the service that implements the features. If you're going to read a file out of Google cloud storage or you're going to try and read your schedule out of Google calendar, there's a variety of things that have to happen when somebody outside of Google calls a service that runs inside of Google.

You have to authenticate who they are in this. There's a lot of different ways things can be authenticated based on the context from which they call. You have to enforce policy on that traffic, make sure that a particular consumer isn't exceeding their quota, or exceeding the amount of money they are able to pay for a service.

You have to ensure that they're confirming with the terms of service and there can be some variety there. They have to sign the terms of service in many cases. A lot of these cross-cutting concerns that need to be implemented in this proxy layer before the traffic actually makes its way all the way to the service backend.

You want this to live in the proxy, because you don't want all of that code living down in each individual service. It's a lot of behavior and it can change quite rapidly and the service behavior should be decoupled fro that life cycle.

It's very common in API management to use proxies to do these types of things. Not just inside of Google, but I'm willing to bet hard dollars that every other big company that has an API product is doing something like this the majority of the time. Certainly, all the commercial enterprise, API management products sell proxies.

[0:08:39.7] JM: That term proxy, could you explain what that means and also explain what the term reverse proxy means? Because I think there are some people who are listening who are a little less familiar with these terms.

[0:08:49.9] LR: Okay. Proxy is essentially a process that receives traffic from the network and forwards that traffic to somewhere else after having done some work on the traffic. It's basically just an intermediary in the network. There could be a variety of different types of proxies. Mostly the types of proxies are named after the network topologies in which they work. Also to some degree, how they work.

A reverse proxy is generating a proxy that receives request for lots and lots of different services. Then forwards that traffic to lots of individual services internally. All the clients talk to one proxy instance, or basically replicas of that proxy instance. Then those proxy instances talk to lots of distinct things internally.

From the outside perspective, proxy looks like it's in different things when in fact it's just one actual physical thing. Internally, all the traffic is read to distinct services. There are other types of proxies; forwarding proxies, gateways and etc., that have slightly different topologies.

[0:09:58.8] JM: In your talk, you were talking about this API layer and you gave some milestones in how things changed at Google in response to changes in the external world. For example, when smartphones came around, it really changed the amount and the type of traffic that you were receiving at that API layer. How did the introduction of the smartphone changed the API topology?

[0:10:30.4] LR: In the world before smartphones, most API traffic was driven by dynamic webpages. We all remember AJAX. You know that. You tended to have, let's say a part of Gmail, it would have its own API that was consumed by the UI. They were very closely coupled and their API weren't – they were effectively in the same trust domain. They didn't need an awful lot of functionality in terms of those cross-cutting behaviors I mentioned earlier.

Then you had API that were designed for public consumption, but there weren't that many of them. They were based on the products that were – the web available products. There was an API for important contacts into Gmail, or reading information at a calendar.

When mobile came around, or you had basically a vast explosion in client experiences all packaged up in these things called mobile applications. Now you had a mobile application which is a crossword puzzle, or a mobile application which is – there's so many examples, it's almost hard to track. Your banking application. The way those mobile applications would work, they would store some state on the device, but they also needed to access services on the backend. This precipitated a massive explosion in the need to create APIs that were effectively externally facing, because they now had a slightly different security model than traditional web-based APIs. They looked much more customer-facing APIs, and in many cases they were, because sometimes an application would use APIs from multiple different vendors.

These APIs needed all these facilities around quotas and rate limiting and security. What really happened was the vast number of them and some of them were incredibly popular. They drove huge amounts of traffic. It was this hockey stick moment, not just in how many APIs were being launched to Google, because it was just a huge increase in the rate. Also how much API traffic was being received in total.

It used to be the all the API traffic we receive was because of small enterprise integrations for e-mail, calendar back off as apps, or productivity apps. Now you have people playing games on these games need APIs and there's a 100 million people doing it.

[0:12:59.9] JM: The increase in the volume of the – Basically the number of different APIs you were serving, that sounds like a problem of API management. It doesn't necessarily sound like the traffic volume is necessarily any different than the type of traffic volume that you might be handling from just the search API. I mean, the volume going to the search API was significant. It's just that now you had this explosion in the cardinality of APIs. Am I understanding things correctly?

[0:13:30.8] LR: I think we had a bit of both. We absolutely had an explosion in the cardinality. There was also something of an explosion in the amount of traffic. What really happened is the traffic shifted from rendering webpages to serving APIs. Any infrastructural investments that had occurred in massive scale web serving shifted over in two massive scale API serving. There wasn't orders of magnitude increase in the net amount of traffic for any particular functional area, but it shifted in its dynamic.

[0:14:05.1] JM: In this new world with the increase in cardinality and perhaps increase in volume, you did build some supporting technologies inside of Google By the way, this was like what? 2006 or 2007, sometime around then, or earlier?

[0:14:22.0] LR: I do believe in 2007. I worked on social networking for about 18 months. Then I started to work on API infrastructure. This was between 2007 and 2010.

[0:14:32.6] JM: Okay. Got it. What were the types of supporting technologies you built for this change in environment?

[0:14:40.8] LR: It used to be that people when they build APIs at Google, they were just embedding libraries in their services. The first transition was getting people over to the proxy model I just discussed, where there was a team that built an API management proxy and then gave all these APIs these cross-cutting behaviors and that allowed more APIs to launch faster. That was really just facilitating the acceleration of the API launch process.

[0:15:10.6] JM: I believe you eventually built a control plane. Can you explain what a control plane is?

[0:15:17.1] LR: In API management, there's really two things going on when you receive an API call. One is any transformations or things you need to do with the data that's in transit. You may need to receive an API call that's in one form. The internal service is expecting a different form, you have to do the conversion and then you send the data along.

Then as a separate parallel set of concerns, which is what are all the policies that I want to enforce in this traffic? How do I extract telemetry from it? How do I report that telemetry? You can actually decouple those two things. That's what we did. We created a proxy, which is the data planning, and then we have a control plane, which implements all the policies.

The proxy would just simply ask the control plane, now what should I do when I receive this API call? Should I let it go through? Should I reject it? Should I do some subtle combination of those two things? There's a fair amount of complexity in those things. Essentially, we were separating those two concerns out. The primary reason why we separated those concerns was for stability and performance reasons.

Proxies work better if they're really dumb. The less work they have to do, the faster they can go. The more you can take out of the proxy, the more traffic you can show through the proxy. Being able to separate those two things and scale them differently allowed us to have a better resource profile to serve traffic faster, and also to reason about the behavior of the system a little better.

It also allowed us to accommodate some additional use cases in particular when you have API calls that lead to long running operations, you may still want to enforce policy as those operations continue.

[0:17:00.2] JM: Just as like we've explored in other areas like big data, for example. Google was tackling the same problems that the rest of the industry is tackling, I guess 10 years or in this case, something like 7 years, or 8 years before they became open source and proliferate throughout the rest of the industry. We're going to eventually get into discussing Istio, which the model of which looks very similar to this control plane and data plane dichotomy that you're describing.

There's a few more historical moments that informed how your own view of API management shaped the development of Istio that we should explore a little bit. The first shift was the mobile change. Then as you said at this talk at QCon, "The cloud happened." What does that mean?

[0:18:05.6] LR: While mobile traffic or mobile revolution, I guess dramatically increased the number of APIs and did shift some traffic away from the web, cloud really represents a massive shift in the amount of traffic.

Now you have clients running on VMs that are on the same physical network effectively as the services they're calling when they're calling a different vendor like Google. They're not just like, "Let me get my contacts at a calendar now." It's your memcache, it's your big table. It's these low-level infrastructure APIs that have to run at massive scale with very low latencies. Applications are extremely chatty when they call up.

You're talking about APIs that are consumed two, three, four, five orders of magnitude more than a typical web or integration API back with. That scale was radically different than what we were had been serving with the prior stack. While the prior stack worked recently well at the scale that it was targeted at, now you end up in a situation where you have some class of APIs that are maybe serving hundreds of requests a second.

Then you have other classes of APIs that are reserving millions of requests a second. It does not make any sense whatsoever to have those two things impact each other. You want to separate them, because they have extremely different scaling requirements. You also don't want a failure mode in one API to impact all the other APIs.

Going through a centralized proxy, you start to get those types of effects. You have, the proxy becomes a single point of failure and that single point of failure affects APIs regardless of what their traffic patterns are and that's not what you want.

We wanted the ability to be able to say, "Look, on a service by service basis, we want to be able to decouple them from all the other APIs." We moved into a sidecar model, where basically each service gets its own proxy. Those proxies are actually co-located with the service itself, so they scale horizontally with the service that they're not impacted by a proxy failing on some other service. There's a much stronger layer of separation. Also by adopting the sidecar model, we get a better network path topology into the service itself.

[SPONSOR MESSAGE]

[0:20:33.1] JM: This episode of Software Engineering Daily is sponsored by Datadog, Datadog integrates seamlessly with container technologies like Docker and Kubernetes, so you can monitor your entire container cluster in real-time.

See across all of your servers, containers, apps and services in one place with powerful visualizations, sophisticated alerting, distributed tracing and APM. Start monitoring your microservices today with a free trial. As a bonus, Datadog will send you a free t-shirt. You can get both of those things by going to softwareengineeringdaily.com/datadog. That's softwareengineeringdaily.com/datadog.

Thank you, Datadog.

[INTERVIEW CONTINUED]

[0:21:22.6] JM: The V1 model. Before you moved to this sidecar model was traffic comes in, it gets load-balanced across different instances of a service. The proxying happens where exactly? Where do the proxy happen in the V1 model?

[0:21:40.9] LR: Well, in the V1 model, which was the thing before I started working on it, all these was done in library code running inside the service itself. In some ways, the V1 model and the V3 model are much similar, much more similar than the middle one. There's no centralized proxy. All the traffic goes from the client effectively direct to the actual service itself. Or there may be other proxies sitting above in the stack, but they're more generic infrastructure than API-specific proxies.

[0:22:13.5] JM: In the V2, where does the proxy sit exactly?

[0:22:17.4] LR: In V2, the proxy's a middle proxy. A client call this, so it's one network hub and then there's another network hub where it goes to the service background.

[0:22:27.7] JM: I see. Is that basically a load balancer?

[0:22:32.4] LR: Yes. Proxies are very frequently load balancers.

[0:22:35.4] JM: I see.

[0:22:36.5] LR: They almost always are load balancers.

[0:22:38.7] JM: Okay. I see. This proxy was a single point, I feel like. Actually, we had a previous show recently about load balancers and this seems to be what happens is they become single points of failure if you're not careful. You are able to negate this by moving the proxying layer into the sidecar model where each – I think you can do this two ways.

Well, I guess do you have a sidecar for each instance of a service, or do you have a single sidecar for – I guess, you have to have a sidecar for each instance of the service, right?

[0:23:12.5] LR: Yeah. For each instance. If we have a service and it's running 10,000 jobs, it has 10,000 sidecars.
[0:23:20.1] JM: Yes. Then this control plane API, does that also have redundancy built into it?

[0:23:27.6] LR: That's redundant, because its needs are very well codified behind an API. What we do have is that the sidecar proxies are able to continue functioning if there are intermittent failures when calling the control plane.

We try to codify those failure modes into the API, so that the sidecar proxy can either have degraded but still functional behavior, or a continuing functioning behavior, even if the control plane is temporarily unavailable.

[0:23:59.6] JM: As you move to this control plane – sorry, this post-cloud model with the sidecar, how were the protocols changing around this time? I know, Google's had protocol buffers. I think GRPC was developed eventually. What were the changes at the protocol layer that were happening around these changes in traffic volume?

[0:24:25.1] LR: Up until cloud standards, JSON REST, or even REST and XML would probably meet most people's performance requirements. At least, it met the requirements of Google's public-facing APIs.

As soon as you're using an API to talk to something like memcache or big table, or really any storage system, or systems that have latencies close to memory latencies, then JSON parsing actually becomes a very noticeable component of the observed co-latency.

Also wire density starts the matter quite significantly. JSON is not a particularly efficient representation of data on the wire, even with compression. Compression also contributes to latency. There's a variety of concerns there.

Google at the time had decided to invest in building out to your PC to facilitate launching these infrastructural APIs that we were building. Also helping codify some interesting development models and some API use cases that were hard to do with traditional JSON and REST.

Aside from that, the basic efficiencies and latency improvements you got with using GRPC and Protobuf. GRPC for instance also supports streaming, and not just unit-directional streaming but bi-directional streaming, which can be a big efficiency win for certain classes of workloads. Google had committed to building out a number of services with GRPC. We needed the API infrastructure to be able to handle that. Our API proxies fairly quickly have the GRPC support, because GRPC is effectively just an evolution internally of Stubby, and the API proxy was already converting JSON REST calls into GRPC, or Stubby calls depending on what the internal system was using at its protocol there.

We build that for instance technology that makes it easy to convert GRPC calls into JSON and REST calls and build that into the proxy. That we could give people two different flavors of the same API automatically. It's actually something that we've open sourced.

[0:26:43.5] JM: Around this time, this was when Google's cloud business started to develop, I believe. There was what you call a convergence of the internal systems and the cloud. Security and network and reliability, I guess the concerns that you would need for the cloud business began to overlap with what you would need for the internal business. Describe what was happening.

[0:27:10.6] LR: Right. When your clients are on the same physical network as the services that you're offering them, obviously they effectively have the same latency performance requirements as the internals consumers, those same services. There was this natural effect of convergence of in what people's expectations are of the system.

We want to facilitate external clients having the same features and benefits that internal clients have. We want to make sure that it's easy for them to call the services that they perform as well, as internal used cases are able to consume those services. That they have the same security and authentication, authorization, abused provincial models.

There's this natural convergence, because Google had built solutions internally to help solve those problems for its own use cases, where we have internal clients calling internal services. We just want to naturally bring those other features to market. Whereas, before we had just been throwing services over the wall.

There was this performance and scale convergence, a convergence in operations management and also in how people think about securing services, how they think about controlling access to data, all those different types of concerns. The trends we see in the cloud industry have very much been along the lines of, we're basically, us and all the vendors building services that map features that we had built out for internal infrastructure.

[0:28:45.2] JM: As these shifts that occurred in mobile and cloud increased the volume and the number of services that you had running, I think you pointed out in your talk that it became a challenge in it of itself to have Google get an understanding of what its own network topology was, to be able to observe and visualize what's going on in the network. How does that discussion relate to the discussion of proxying and API management?

[0:29:17.7] LR: Yeah. I mean, Google has a lot of infrastructure that we're pretty disciplined internally in telling teams – in teams using this infrastructure to build our services. We have a lot of uniformity. We all use the same libraries to build servers, to build clients, to describe how servers and clients are supposed to talk to each other, do load balancing and those types of things.

It was still a challenge with any Google to understand the topology of systems, to bill or trace calls, but we had built our tools that had done a lot of those things for us. You've seen a lot of those tools come to market. I mean, Google published I guess the Docker whitepaper many years ago. Now there is Zipkin and OpenTracing and Wide Step and all those types of companies.

We had done a reasonable job with those tools, enough to make ourselves fairly happy internally. What was more of a challenge when we had cloud customers now basically consuming our infrastructure, they need the same types of insights and they need insights that span not only their services, the ones that they built and consume for themselves, but also mixtures of consuming their own services and consuming our services.

Being able to provide people a holistic view over their service dependency topology, when that topology spans things running on premise, things running on cloud, services provided by a cloud provider, all right things get a little bit more complicated.

[0:30:50.8] JM: I think we've given people an understanding of the historical context in which Istio was developed. You had all these experience with the API management of Google in those days of burgeoning traffic and – increases in traffic, I should say. Now, people that are developing their own companies on infrastructure that looks like that of Google's, stuff like Kubernetes, they have similar needs. They are getting an expanse of lots of different services that they're working with. There are things that they want out of every service.

As you point out in this talk, if you're not careful you can end up just bundling all of these application logic into your actual app, and that's a lot more cumbersome than standardizing on what you want out of every service and putting it in a sidecar. The sidecar pattern is often implemented with a container, so you have a container that sits alongside your services container and it does these things that every service wants. What do we want out of every single service? What are the standards that we want to have?

[0:32:10.8] LR: Right. If you think, microservices aside in any endeavor where people are billing services that are talking to each other, there's a few kind of standard things that you probably want the network to take care of for you.

Let’s take HTP as an example here, because a lot of people write services to service communication using things built on top of HTP. You probably want something that’s going to handle retries for you. If a service returns temporarily unavailable, or service unavailable you probably want that traffic HTP request to be tried somewhere else, or tried again.

Retry logic is a very common behavior that people end up writing in code. It’s something that the network could quite easily do for them. Then also, you might want to write a piece of code that says, “Well, if this HTP call is taking too long, I want it to stop. Because I need to go do other things, or I need to serve some information back to the user.” I can’t make them wait forever, so we need to have some limits. We want a timeout. Again, in a very common bid here that’s written at libraries.

There’s a longer list of these things as topology gets more complex, or you want more load balancing features, or become higher traffic, you’ll want load balancing features, you might want circuit breaking. If you’re running a big system, you might want to do chaos testing, where I’m going to deliberately inject holes into my network and see if my application continues to function.

Again, these are things that people do build into libraries. That really starts to pile up. It really, really piles up when you have lots of application development teams all writing in different languages, all implementing those same essential features in a different way. Because there are plenty of examples in SRE lore of poorly implemented, retry logic, or timeout logic causing thundering hard problems.

[0:34:06.0] JM: One thing you didn’t mention there, but you had spent a lot of time on this in your QCon talk was security. What do we want in out of every service in terms of security?

[0:34:16.7] LR: I think when services or workloads talk to each other, they want a couple of different things. One, the server wants to know who’s talking to it. At least some idea of who’s talking to it, because it probably wants to enforce policies.

Conversely, the client wants to know that the server its talking to is actually the server it thinks it’s talking to. You need this mutual trust model to fulfill this requirement, at least speaking specifically about workloads within the data center. Because if you don't have some form of mutual trust.

I mean, if a client is talking to a server, it's possible that somebody may have brought up a rogue server registered in the service registry and now the client is talking to what it thinks is the real server and sending it privilege user information, when in fact it's scraping all that information and sending it to some nefarious entity.

Conversely, the server wants to know – this is a client – it's a client whose identity I can validate. That identity is allowed to perform certain tasks and model out to perform and serve other ones. It would be very nice for application developers if they didn't have to build all of that themselves over and over and over again, because it can actually be quite complicated. If you get it wrong, then you have gaping security holes and there have been many notable examples of that over the years.

Effectively, you like the network and the workload orchestration system like Kubernetes to help you with these problems. You'd like the workload orchestrator to basically when it runs workloads, be able to give them identities that they can use when they communicate with other services.

These are the properties that I think you want. You want the network to basically enforce those identities are properly propagated and validated. You want their orchestrator to make sure that workloads have identities and that those identities can be used with the network.

[SPONSOR MESSAGE]

[0:36:19.4] JM: If you are building a product for software engineers, or you are hiring software engineers, Software Engineering Daily is accepting sponsorships for 2018. Send me an e-mail jeff@softwareengineeringdaily.com if you're interested.

With 23,000 people listening Monday through Friday and the content being fairly selective for a technical listener, Software Engineering Daily is a great way to reach top engineers. I know that the listeners of Software Engineering Daily are great engineers, because I talk to them all the time. I hear from CTOs, CEOs, directors of engineers who listen to the show regularly. I also hear about many newer, hungry software engineers who are looking to level up quickly and prove themselves.

To find out more about sponsoring the show, you can send me an e-mail or tell your marketing director to send me an e-mail jeff@softwareengineeringdaily.com. If you're a listener to the show, thank you so much for supporting it through your audienceship. That is quite enough, but if you're interested in taking your support of the show to the next level, then look at sponsoring the show through your company.

Send me an e-mail at jeff@softwareengineeringdaily.com. Thank you.

[INTERVIEW CONTINUED]

[0:37:46.9] JM: I think, people who want to know more about implementing this – implementing Istio in Kubernetes deployment can obviously listen to the lasts episode that we did. Just to put some of that into this episode, can you describe the relationship between – we've got a control plane. We've got this control plane API that has certain functions that it provides and then we have a sidecar proxy that sits alongside each instance of our service and it does certain things. Can you just give an overview of the relationship between the sidecars and the control plane API?
[0:38:27.0] LR: Right. The control plane API is really the thing that tells Istio what you want the behavior of your network to be; how you want to route traffic, what policies you want to enforce in that traffic.

The control plane API is designed to be a place for both service developers and operators to be able to get their job done. In particular, operators because it gives them a lot of tools that they didn't really have before; where they can influence behavior of traffic in a much more fine-grain way. Or they can impose policies that don't require the application code to change.

Envoy is the sidecar, which really is designed to implement those policies. Let's say the control plane API says, "I want Envoy to round-robin load balance across all the backend for a particular service." Then the control plane API, an operator will write a rule that specifies that.

That rule basically be synthesized by a component and system co-pilot, which will then deliver that configuration down to Envoy, which will then basically make that rule effective.

Envoy is the – as a sidecar is really a logical extension of the application that just happens to sit out of process, that implements the networking behavior that you want. It does that in a few different ways.

One, it obviously affects the routing of traffic, so what we call the data plane; how traffic gets from service A to service B. It also integrates with the policy layer at runtime, so that as traffic goes from A to B that we can check policies on requests, on connections and things like that.

That's really the relationship. The control plane API really is an API that's just focused on enabling operators and developers to control the behavior of a network. If you look at classical STN systems, they have an API that allows you to control the physical topology of the IP network. Well, the control plane API in Istio allows you to control the topology of both the layer 4 network and also the layer 7 network.

[0:40:34.1] JM: One of the aspects of Istio is that it's built in the context of Kubernetes, but it is platform agnostic. Can you explain what that means?

[0:40:44.5] LR: Right. Whenever you run services, you're probably using something to orchestrate how you run them. Kubernetes obviously has done very well in that space, particularly scheduling and orchestrating services to run its containers.

Workloads run in lots of places. They run on my laptop, they run on VMs that are brought up manually. They run on Mesos, they run Docker Swarm. There are lots and lots of ways of running workloads. What we lose off the term 'orchestration systems.'

Most companies of reasonable scale tend to be using multiple of these things. In fact, it's very hard to think of an example customer where they aren't using more than one of these things. Istio needs to be able to work in a world where the workloads are orchestrated by lots of different systems, and those systems also often control how services are registered for things like DNS, or console, or other endpoint discovery systems.

Also where their services span distinct, physical network or different IP name spaces. Istio has to be able to stitch all of those things together. While we did ship Istio a fully functional version of Istio working with Kubernetes first, because it had such great traction in the market and was a fairly easy platform to integrate with, we don't require it.

It's totally fine to use Istio or all your workloads running on VMs, or where all your workloads are running, scheduled by a nomad, or where all your workloads are running on hardware sitting on premises. Istio actually doesn't really care how the workload is run, as long as you can meet certain integration requirements.

[0:42:33.2] JM: That agnosticism extends to the things that people can integrate with Istio. If I want to have a monitoring system, I can use Prometheus. I could plug in Prometheus. Prometheus actually comes with it by default, but it's easy to swap out Prometheus with other stuff, like the same goes for Zipkin as my distributed tracing tool of choice. I could swap it out with other distributed tracing tools of choice. Can you describe the APIs that you want to expose within Istio and where you want to expose those, so that people can mix and match the different plugins for their operations?

[0:43:17.9] LR: Right. If we have this baseline of sidecar proxies are able to run everywhere associated with their workloads, what those proxies do is they extract information from the traffic as it close the network. They call a component called Mixer, which enforces both policy checking to gate weather that traffic is allowed to proceed or not. Also, to receive telemetry, which it can then forward to telemetry collection systems like Prometheus.

Mixer is the standardized integration point in Istio for integrations. If you want to write a plugin that integrates with the active directory to do [inaudible 0:43:58.9] checking, then you can do that and you should do that in Mixer. If you want to send your telemetry to Datadog instead of Prometheus, you can do it there.

It's this nice neat integration point that we codify for people and make it easy for them to integrate into, while still allowing them to see an awful lot of information about the totality, the behavior and the network.

What we're doing with that particular project Mixer is working with different vendors to have the right plugins, so that at some point, you as an operator of your company services and just take a bunch of these stuff off the shelf and integrate this and configure it to meet what your particular deployment production requirements are.

[0:44:45.8] JM: The Mixer is this point of integration, and the control plane is gathering this data from the different sidecar instances. How aggressively does the Mixer pull that data from the sidecars, or does the sidecar push the data to the Mixers?

[0:45:06.7] LR: Generally, it pushes. On the case of pure telemetry, the sidecars will tend to batch data for a period of time and then report it, just for efficiency reasons. Policy checks – if you don't want an API call to go through, or a server for it to go through –

[0:45:23.6] JM: You want to know now.

[0:45:25.2] LR: You want to know now. You can obviously call Mixer one to one, that's not particularly efficient. We provide ways to help Envoy cache results of those policy decisions. The one thing you can say about policy decisions is very often they repeat. One of the things in our production experience is that services tend to make the same call over and over again very quickly.

Policy checks actually cache very well. We put a lot of effort into making sure that policy checks are cacheable by Envoy, and so that reduces the load on Mixer. It allows us, in some cases for Mixer to do more complicated policy checks. It also makes the system more generally stable. The Mixer is not a single point of failure in the traditional sense. If Mixer goes down, the system will actually keep running for some period of time.

[0:46:17.5] JM: What is the architecture of the control plane API?

[0:46:21.5] LR: It's in some ways very similar to Kubernetes API architecture. We have a standard set of API resources that can figure the different components in the system. They get written into storage by the control plane API and then the different components, the runtime components by the Mixer and the Auth control plane piece are effectively watching the storage system bringing those pieces of configuration into memory and making them live. It's fairly simplistic.

[0:46:51.4] JM: I see. Were there any particular resiliency tools that you had to build from scratch, or were you able to just take stuff off the shelf or recycle things from Kubernetes to maintain the reliability of that control plane?

[0:47:06.3] LR: The control plane itself actually doesn't need to be very reliable. What really matters is the APIs between the sidecars, the control plane runtime components. All right, that's the thing that actually affects availability. There's two types of problems or availability problems; one is you don't need to change the behavior of the system and can you maintain the system at its current behavior without having failures.

Then another class of problem, which is can I change the behavior of the existing system? That's the divide between the control plane API and the runtime. You're usually much more invested in the runtime environment being stable, because that's the thing that's affecting the actual services.

[0:47:53.3] JM: If the control plane is down, the things that maybe won't work so well for me are if one of my sidecars does a check for authentication, okay the request to the mix to the control plane fails, but I can use one of my cached authentications hopefully. I can't send my telemetry data to the control plane API, that's not the end of the world. I can still run and the control plane is down, so I can't send new config out to my proxies, my sidecar proxies, but it's not the end of the world, because my current configuration is probably just fine. The control plane doesn't necessarily need to be insanely reliable.

[0:48:36.7] LR: Right. Maybe, if you look at the standard Istio diagram, we have what we call the control plane API. That's really just the configuration API. That doesn't need to be reliable, because that only affects your ability to change the configuration of the system. It's totally fine for that to be two nines. It's the control plane runtime API that when Envoy talks to Mixer, or when Envoy needs to talk to pilot, that should be much harder than [inaudible 0:49:03.1]. There are things that we do within that, like caching with an Envoy to allow it to go from three nines to four nines to five nines.

[0:49:13.3] JM: I guess, I have a little bit of trouble understanding why wouldn't you make the control plane API more reliable.

[0:49:21.8] LR: We do. I mean, we work very hard in making it as extremely reliable. As you go from systems, they're trying to achieve three nines in reliability to five or six nines of reliability. One thing there to come into play. All about refinement to that.

[0:49:39.5] JM: What does change? Tell me about the architectural differences between a three nine system and a five nine system.

[0:49:46.9] LR: A three nine system, you might classically, I want one of my components to be horizontally scaleable. Instead of running one instance of Mixer, you might run three or five, or 50 or a 100, right? That's pretty easy to do. It might give me up to three nines, it might even get me up to four nines. But it probably won't get me to five, six, or seven, because there are other things that can happen.

The network connectivity between the sidecar and Mixer might have a problem. It might just be a very niched little corner of the network that's having packet loss. Calls are timing out. That's the level of refinement, where having a cache inside Envoy – even that policy check call is only actually physically getting through the Mixer like one in 30 times, for that one proxy on that one instance of that service, you're still able to rely on the cache to deliver the policy results.

Because when you're up until you – people often measure system reliability in terms of units of time, if you're up into four nines, you're talking about minutes and seconds out of a year of about available downtime. If you had let's say 10 jobs running a particular service and one of them loses network connectivity for 10 minutes, but you could start to violate your SLA. It's availability in-depth. You have to do all the things.

[0:51:18.3] JM: Okay. I know we're up against time. In this talk, you gave these two milestones; the mobile traffic and then the changing cloud traffic. Since that time, Google has really been a little more open with at least some of their technologies in terms of how aggressively they've gone into open source and then obviously there is lots of managed services. Is there anything that you're seeing from the inside, where you're like, "Oh, this is going to be the next big source of crazy traffic that's going to change the API layer."

Maybe one thing that comes to mind is just the stuff that can go on in distributed machine learning systems, that almost seems like – whereas, with something like mobile, it's like at least you're gated by how aggressive the external users are. With something like distributed machine learning, if Google says, "You know, we've got these machine learning jobs. If we just have extra resources, we can allocate those resources to machine learning jobs. That means that the demands that your system are just going to scale infinitely." Are there any upcoming challenges to the API infrastructure that you're seeing on the horizon?

[0:52:29.8] LR: I mean, there are certainly are – machine learning is its own special world. Almost where you have lots and lots of these – forget about microservices or nanoservices. These TPUs, which are computational units, which make calls between each other on physical hardware that's dedicated towards that. Running a sidecar proxy between those two things would make absolutely no sense.

At a certain scale and a certain latency requirement, proxies don't make any sense. Now you really have to push functionality all the way down as far as you humanly can into the application layer. Things start to get pretty crazy.

At a certain scale, you're probably going to want to use very, very dedicated, even hardware dedicated libraries to do certain classes of things. I think the sidecar model is going to actually scale it very well for the overwhelming majority of typical enterprise and average storage use cases. That's a pretty long runway frankly.

I think what you'll see is just a segregation in between this kind of 90th – the 90% of workloads which don't need dedicated hardware. The 10% of workloads that do, but the 10% of workloads might represent money percent of the trial.

If you look at what's going on in machine learning and people building dedicated harbor for that, I mean it's pretty clear. There's just no way without building for the hardware that you're going to be able to compete.

[0:54:08.9] JM: By the way, does that stuff start to get difficult for you? I did one show about distributed deep learning, and it was the hardest – probably the hardest show that I've done. I just got so confused so fast. I mean, you're a principal staff engineer I think, but so I don't know. Do your skills start to be tested in that domain?

[0:54:28.7] LR: Yeah. I won't say my eyes glazed over, but I think go and read a white paper that's published internally that describes how some of the stuff might work. I get at the high-level, but it's a bit like reading a brief history of time. Yeah, it's a great read. This all totally makes sense to me, and then you give me the post-graduate masters thesis textbooks on quantum mechanics and I've lost the brain.

There is just a cliff of specialization that occurs and it definitely internally too. Yeah, there are limitations to where, I think Clint Eastwood put it best. A man's got to recognize his limitations. That definitely applies to software development too. While I think Istio will be very useful for a lot of people, there are definitely some people who I'm not pitching Istio to for a good reason.

[0:55:21.0] JM: Got it. All right, Louis well it's been a pleasure talking to you. Thanks for making time at the end of a Friday. I really enjoyed your talk at QCon. I recommend people check it out. I'll put it in the show notes. I'm really excited about Istio and I plan to continue reporting on it astutely.

[0:55:37.0] LR: All right. Well, thanks for having me on. It's all had been fun. Yeah, hopefully people find the QCon material interesting and engage with the community, try it out and hopefully it does them some good.

[END OF INTERVIEW]

[0:55:51.8] 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 visualize them end to end with the value stream map. You get complete visibility into and control over your company's deployments.

At gocd.org/sedaily, find out how to bring continuous delivery to your teams. Say goodbye to deployment panic and hello to consistent predictable deliveries. Visit gocd.org/sedaily to learn more about GoCD. Commercial support and enterprise add-ons, including disaster recovery are available. Thanks to GoCD for being a continued sponsor of Software Engineering Daily.

[END]