EPISODE 500

[INTRODUCTION]

[0:00:00.7] JM: Oliver Gould worked at Twitter from 2010 to 2014. Twitter's popularity was taking off and the engineering team was learning how to scale the product. During that time, Twitter adopted Apache Mesos and began breaking up its monolithic architecture into different services. As more and more services were deployed, engineers at Twitter decided to standardize communications between those services with a tool called a service proxy.

A service proxy provides each service with features that every service would want; load-balancing, routing, service discovery, retries observability. It turns out that lots of other companies wanted this service proxy technology as well, which is why Oliver left Twitter to start Buoyant, a company that was focused on developing software around the service proxy and eventually the service mesh. If you're unfamiliar with service proxies and service mesh, you can check out our previous shows on Linkerd, Envoy and Istio.

In this episode we're going to be going a little bit deeper than our previous episodes. Kubernetes is often deployed with the service mesh. A service mesh consists of two parts, the data plane and the control plane. The data plane refers to the sidecar containers that are deployed to each of your Kubernetes application pods. Each sidecar has a service proxy. The control plane refers to a central service that aggregates data from across the data plane and can send communications to the service proxies sitting across the control plane. So it's really bidirectional communication.

You've got all these different pods in your Kubernetes cluster. Each pod has a service proxy sidecar container. The sidecar container is communicating with that centralized service mesh. It's communicating what's going on in this pod, and then you've got this centralized service mesh which can be used to deploy updates to the service proxies. Maybe you have a change in your configuration or your Policy Management, and the service mesh can be used as the communication point to communicate that out to all of your different pods in the distributed cluster.

The first product that came out of Oliver's company, Buoyant, was the Linkerd service mesh, and that was built in Java. The project was started before Kubernetes had become the standard for a container orchestration. So Linkerd was useful for connecting a service mesh through Mesos and Kubernetes and Docker swarm. It was very flexible, but once Kubernetes became the standard for container orchestration, Buoyant decided to build a new service mesh, which is called Conduit, and this is a service mesh built using Rust and Go.

I think this is a great example of why it's so important that we eventually centralized on some kind of container orchestration system rather than all of the different companies in the space using either Mesos or Kubernetes or Docker swarm, and yes, there are people who are using those other container orchestration systems, but the centralization towards Kubernetes allowed for, in this case, Buoyant to build a service mesh specifically for Kubernetes and it's built using Rust and Go, so it's a lot faster and lighter weights than Linkerd.

In this episode we explore how to design a service mesh and what Oliver learned in his experiences building Linkerd and Conduit.

[SPONSOR MESSAGE]

[0:03:40.9] JM: Azure Container Service 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 into any one vendor or resource. You can continue to work with the tools that you already know, such as 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 off-line. Isolate your application from infrastructure failures and transparently scale the underlying infrastructure to meet growing demands, 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, and the link is in the show notes. Thank you to Azure Container Service for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:05:07.3] JM: Oliver Gould is the cofounder and CTO at Buoyant. Welcome to Software Engineering Daily.

[0:05:12.8] OG: Thanks, Jeff.

[0:05:13.6] JM: We've done a few shows with your cofounder, William Morgan, and the first one we did with him was a discussion of Twitter and his experiences there, and you spent a parallel period of time with him at Twitter. To set the stage for a discussion of modern infrastructure, let's reflect on how things were just seven years ago at a place like Twitter. You worked on traffic and observability during your tenure, which was, I guess, 2010 to 2015. That was a time when traffic was increasing, observability was increasing in importance. Can you describe your experiences?

[0:05:54.6] OG: Oh, yeah. That's a great question. When I joined in 2010, I was coming from Yahoo and Yahoo is this very large millions machines all over the world, but a very kind of messy internal architecture, and I was really focused on config management at Yahoo. So coming into Twitter, I expected to be like, "Okay. I will be working on puppet," and we recently had a lot of puppet and all of that, but quickly I was kind of thrown into what I now know to be micro-services migration.

We didn't have those words for it then. I think we called it system oriented architecture or one of those bad words, but I was quickly kind of thrown into the re-architecture that Twitter build into, and that was having observability as a service, having compute and memory as a service via the memcache team and the Mesos team having kind of rich traffic libraries and Finagle, having your kernel team and really having a distributed operating system for Twitter engineering to build into on the platform teams that Twitter build into.

That was very lucky time to be there, to be there when it was, I think, 30 engineers and run-time systems when I started and then 150 two years later. It's just really lucky to be behind the scenes and see what actually goes into making these things work and go to production.

[0:07:07.7] JM: So back then, you were on Masos. Twitter I think is still on Mesos or —

[0:07:14.6] OG: Yeah. When we started working on observability, the team next to me was the Mesos team and they were our team and I totally bet against them. I thought that they were going to fail and nobody would ever give up ownership of their own hosts to like run in some scheduled environment. We really rebuilt observability not expecting the dynamic environment to win, and about a year later we really had to scramble to really meet the mass adoption, because Mesos unlocked so much flexibility in engineering. You no longer had to go ask someone for a piece of hardware and pay money, you could just go launch your process in some big harder pool. That was surprising. It was nice to be wrong sometimes, I guess.

[0:07:54.7] JM: I imagine that introduced all kinds of observability implementation differences, because you were going from — I guess, that was a world where people were just ad hoc spinning up VM's that were not on some centralized addressable platform. It was going from what to Mesos.

[0:08:13.8] OG: Oh, yeah. Initially we had kind of a host database like everyone starts with, just like a machine catalog. To build observability, since I was in the ops team, I know who to go ask to get 30 machines for our infrastructure and we'd go manually. SCP or our binaries or Capistrano deploy your binaries on the those fixed data coasts, and so kind of no dynamic environment, rented data centers, our own data centers and a host where I could go and I knew the name of it and I could go find it in the data center if I wanted to.

That was kind of the early days, and everyone would do that. It was very manual and the way we do service discovery, it was initially by like just copying around host lists between teams, and that clearly wasn't going to scale with the engineering team and the velocity that we need to operate at.

[0:09:02.5] JM: I think I'm remembering this now. I think William told me there was like a shared Google spreadsheet.

[0:09:07.7] OG: Yeah. That's basically right. If you knew like what whiskey the hardware division team like, you could get your hardware a little easier than other people. It's very manual back then.

[0:09:17.0] JM: Was there any hope of standardizing observability tools across those different host instances?

[0:09:24.3] OG: The other thing to think about here is that this is kind of before the big micro-service decomposition. So there were few of the kind of platform services that have been extracted out of the monorail, but this is before there were real distributed micro-service. It was really one big application being deployed every day, and so observability was mostly done to Ganglia and Nagios and was basically statically configured and it was basically okay, but there is some smart people in engineering who realized that we'd have to have a much more distributed dynamic environment to actually scale the engineering team, that like we couldn't keep adding people to this one project and have it be at all maintainable.

That's kind of where the need for an observability platform came from, and when we started with the observability platform, it was still kind of you'd file a ticket to us and we'd start to monitor your static list of hosts that we could go pull from the database or from the spreadsheet. Over time, that became more and more automated to where now in their Aurora clusters, as you deploy, you just get visibility for free just by virtue of getting deployed in the cluster.

[0:10:27.8] JM: This move to having an increase in the number of services, I guess that coincided with Mesos. Either Mesos — People wanted to have Mesos because there were more services and it was becoming a little more of a sprawling architecture, or the sprawling architecture was happening because of Mesos. Maybe you could describe a little bit about that time as people were onboarding to Mesosand the monorail was getting broken up.

[0:11:00.3] OG: Yeah. As you can imagine, Twitter at the time had a lot of tech debt in platform that we were trying to pay off and at the same time I think Twitter probably still has this problem.

They're focusing on growth. How do they add more users to the product month over month? To that, they had to be able to iterate much more quickly on the product, and so doing that in one big Ruby application really wasn't going to fly. There was a time where people were just trying to jam all these features into the main application, and then as that became organizationally difficult and would slow things down, it kind of became natural to want to create new services as a way to just own your own schedule.

So William, for instance, launched the photo service. If you see a picture on Twitter, it's his fault. Then that was one of the first decompose services and that was kind of done, so they could launch quickly without having to be fully integrated into the main application. It's kind of that — Then it got harder and harder to get your own hardware, and so there is this kind of tension that push people in a Mesos as the only platform where you could really move quickly, and micro-services has just become kind of a cultural thing that fell out of that.

[0:12:12.7] JM: And Finagle eventually was a project that got started, if I recall correctly, to standardize some communications between these different services. As the services proliferated, people started be saying, "Well, we should have some standard model for proxying communications between these different services. Let's build a thing for it."

[0:12:35.2] OG: Yeah. It'd be nice if it was that intentional. In some ways it was. I think management really funded it in that way. But it started actually as a failed attempt to write a reverse proxy, and so they were going to write the Twitter frontend initially. I think this is mid 2010 or so. As they started to go through this, they realized they actually had to like develop all of these kind of core pieces to write it. So they were building over Nete and they were writing and Scala and they just realized they needed all these new abstractions to really make that ergonomic incorrect. That project can fail. The engineers working on it realized that they just solved a bunch of problems that every other engineer at the company was solving every day.

I was working on a few other projects at the time on the observability stack and I was solving services covered in load-balancing and retries and timeouts and all of that stuff. Another engineer on my same team was in a slightly different stack and had to solve all the same problems differently. So that's kind of when the lights clicked to me, like, "Oh Why don't we just all pile on this one library and try to make it work for ourselves?"

Eventually that got funded into its own team, the Finagle team or the CSL team at Twitter now, and yeah, it's been nice. It has been funded as kind of a whole layer of [inaudible 0:13:53.4] infrastructure, is this core library.

[0:13:56.4] JM: So this is like a platform engineering team got started at Twitter.

[0:14:00.2] OG: It is. Yeah, it is. It's always been very closely related to the platform engineering team.

[0:14:06.1] JM: I want to make our way towards Buoyant and Linkerd and then to Conduit pretty quickly, because we've done a bunch of other shows about service meshes and we've done shows on the history of Twitter. We should get into the present in the future. Maybe you could just bring us through what happened when you and William left Twitter and you started Buoyant, which is a company that is building these service meshes. You started out at Buoyant building Linkerd, which is one service proxy kind of evolved into a service mesh technology. I guess bring us up to speed, get us to the point where you had Linkerd and you were thinking about what to do next.

[0:14:50.5] OG: Yeah. So when we started Buoyant, we kind of knew that Finagle and all these kind of platform value at Twitter needed to make it to the outside world. If Kubernetes and Docker and all of these things are actually going to be successful in organizations, because there's just so many of these pieces you have to actually build up to have a same platform. We didn't think everyone wanted to work in the libraries that we worked in,, and so putting it into a proxy was our attempt to encapsulate the operational value that we thought we did really well at Twitter without having to take on any of that programmer overhead of deciding on a programming model or anything like that, which is we don't want to have a holy war of what the correct thing is, we just want to make it work better today. That's where the idea for the proxy started.

As we saw the proxy actually get deploy to play places, that need for a control plane became more and more apparent, and so we have Namerd which is our kind of first attempt to do that. Then as we started to think about what went well with Linkerd, and that was certainly kind of

establishing this functionality of a service discovery where super ops rich proxy with this idea of a control plane being part of it, and we realized that if we were to start over again, we would've started with a control plan, and that shouldn't be an afterthought. In some ways, the proxy is an implementation detail of the control features.

The other kind of big lesson from Linkerd is that the cost of the JVM obviously is fine in many organizations, but especially when you're talking about people in low resource environments running Kubernetes example clusters or startups or especially in the pod side curb model where you have maybe hundreds of sidecar ion a host, the JVM is just not a suitable runtime environment for that.

We knew that going into it, we just didn't know that the architecture would push us into this kind of pod sidecar environment, and we had always been looking at Rust as kind of the ideal solution of what we'd like to build in, but when we started Buoyant, we looked at the ecosystem and there is barely a sync networking implemented and it looked like we'd have to invent the world to even get started. Like I said, our goal at Buoyant is just o provide value for teams who have problems today and not at some distant point in the future when we get a perfect architecture.

[SPONSOR MESSAGE]

[0:17:14.7] JM: DigitalOcean Spaces gives you simple object storage with a beautiful user interface. You need an easy way to host objects like images and videos. Your users need to upload objects like PDFs and music files. DigitalOcean built spaces, because every application uses objects storage. Spaces simplifies object storage with automatic scalability, reliability and low cost. But the user interface takes it over the top.

I've built a lot of web applications and I always use some kind of object storage. The other object storage dashboards that I've used are confusing, they're painful, and they feel like they were built 10 years ago. DigitalOcean Spaces is modern object storage with a modern UI that you will love to use. It's like the UI for Dropbox, but with the pricing of a raw object storage. I almost want to use it like a consumer product.

To try DigitalOcean Spaces, go to do.co/sedaily and get two months of spaces plus a $10 credit to use on any other DigitalOcean products. You get this credit, even if you have been with DigitalOcean for a while. You could spend it on spaces or you could spend it on anything else in DigitalOcean. It's a nice added bonus just for trying out spaces.

The pricing is simple; $5 per month, which includes 250 gigabytes of storage and 1 terabyte of outbound bandwidth. There are no cost per request and additional storage is priced at the lowest rate available. Just a cent per gigabyte transferred and 2 cents per gigabyte stored. There won't be any surprises on your bill.

DigitalOcean simplifies the Cloud. They look for every opportunity to remove friction from a developer's experience. I'm already using DigitalOcean Spaces to host music and video files for a product that I'm building, and I love it. I think you will too. Check it out at do.co/sedaily and get that free $10 credit in addition to two months of spaces for free. That's do.co/sedaily.

[INTERVIEW CONTINUED]

[0:19:34.4] JM: The other thing is that you started Buoyant when we were in the midst of these container orchestration debates. I don't want to use the term war, but you couldn't just say, "Okay. I am just going to build a service proxy system or service mesh," whatever term you want to use, "for Kubernetes." You had to say, "Well, am I going to support swarm? Am I going to support Mesos?" You did say that. You wanted to support Mesos and swarm and Nomad and Kubernetes and all these different things. What was the penalty of the debates around the container orchestration? Was there a lot of work that is looking like it decreased in value, because everybody's moved to Kubernetes now?

[0:20:20.0] OG: In some ways, but we had a lot of lessons out of those interactions, and so Linkerd itself is made to be very generic as an integration tool. You can use it to glue lots of things together. You can glue ZooKeeper and Consul and Kubernetes service discovery. It was always intended to be — It was designed in this way to be very flexible and modular. Honestly, a lot of that design came out of the finagle implementation, and so we weren't adding a lot of extraction overhead to do this. It was more — We knew when to plug in APIs we had to build and just layering them in. But I think it made it much harder to focus on control air features,

because they're very different primitives and features provided by each of these orchestration environments.

When we started to think about Kubernetes as a target for Conduit, we realize that there is a lot you get just out of the Kubernetes API in terms of RBAC and auditing and storage APIs, service discovery. There are so many primitives in there that it just seems if we're going focus in the control plane first and not on this glue level proxy that it works in many environments, then Kubernetes is a really sane bet there in terms of the features and adoption to get there.

[0:21:29.6] JM: And if you were building a system where you can have a service proxy that communicates with something like ZooKeeper, correct me if I'm wrong, but Kubernetes essentially gives you ZooKeeper features out-of-the-box. So that's like — If you're building a service proxy for Kubernetes, you wouldn't really need to — Well, I guess you could need to communicate with ZooKeeper like if you're running a Hadoop cluster that just has to have ZooKeeper within it. Tell me more about that. Once the world has standardized on Kubernetes, which they now have, what kind of precepts can you change?

[0:22:07.3] OG: Yeah. I mean, you'd be surprised. We had worked with a lot of organizations who — Kubernetes is not an entirely greenfield project. They have to integrate this in their existing environments. Maybe there are engineering organizations made a decision months or years ago that curator is going to be a standard in their organization and every application is going to integrate with ZooKeeper via Curator. The move to Kubernetes may — They may have to find a way to keep that immigration going, because it's not an all or nothing move to Kubernetes. It's an incremental move.

We have production users of Linkerd who are using that to glue together their pre-existing curator infrastructure with their forward looking Kubernetes infrastructure. Yeah, Linkerd is a very pragmatic tool. You don't necessarily get to only start with a Kubernetes cluster. You might have to live the old life for little while too.

[0:22:53.9] JM: Okay. We're getting pretty wonkish here, I guess.

[0:22:57.6] OG: Sorry.

[0:22:57.7] JM: No. It's my fault. It's my fault. I should've set the table correctly. The service mesh model, whether we're talking about Linkerd or Conduit or istio, you've got data plane and a control plane, and the data plane is where actual application request traffic between different instances is carried. If we're talking just about Kubernetes, you've got your application pods, and then a pod has a sidecar that is a container within it that is doing the service proxying, the service communications. So every request between different application pods goes through the service proxying container. You've got this container, the sidecar container, that is communicating with any other application sidecar container so that every single communication between a pod and another pod is shuttled through this common component, and that gives you a lot of nice features, because if you can make an assumption that every single communication between different services is going to be seen by this type of container, then you can have some standardizations around the way that these different pods are communicating with each other, and this communication gets aggregated into the control plane. You have the data plane, that is like how all these different instances are communicating with each other, and you've got the control plane that aggregating information from the data plane, and then you can do lots of interesting things in this control plane, because you can change things that are going on in the data plane, because you've just got the centralized point of communication between all of your different pods. Explain a little bit more about that relationship between the control plane and the data plane.

[0:24:50.0] OG: Yeah. That's exactly right. As you said, what we put in the control plane — So where we started with Linkerd was very data plane heavy and we added all the features effectively into the data plane, so service discovery and everything was configured directly into the proxy and only later did we extract that into a different service into the control plane.

With conduit, what we're doing is really starting with the control plane being the very kind of forward part of the product, and so that's written Go. It's implemented as a bunch of GRPC services, and so the idea there is that we'll be able to add a plug-in interface. So if you want to write, for instance, a telemetry interpreter or something that receives latency information and success rate information and all of those things, you'll be able to write just a small GRPC service and drop it into the namespace that Conduit runs in and your process will now start being dispatched telemetry information.

So that's where people build into and that's where your organizational build automation into as a way to both control the proxy's behavior and configure how the proxies behave as well as to receive runtime information from the proxies about traffic is fronting in your system and how the proxies themselves are behaving in relation to each other.

[0:26:05.1] JM: So what kinds of commands would I want to — If I'm an operator and I'm looking at the control plane and I can do all kinds of stuff from this control plane, I can issue commands out to all of these different sidecars that are sitting on all of my pods across Kubernetes. What kinds of commands might I want to issue?

[0:26:24.1] OG: With conduit, we've really started on kind of the observability first features. As we get into, more policy facing features, which we'll talk about in a second. That could be quite a bit more complex and security sensitive. But what we've really tried to focus on with Conduit is, one, not requiring total adoption, that you should be able to add these sidecars, just certain pods or certain services and they don't have to be added to all services for you to get information out of them.

If I only wanted to bug one service, I can just add pods to that — [inaudible 0:26:54.1] to that service. What I can do with control plane is with Conduit we can actually tap into requests, and so you could think about this kind of like TCP dump, but for your inter-service requests, and so it works at the request level and I can say, "Ask the control plane to tap requests that match this path or have these headers," or something like that, and then we can actually select out those requests and send them back to the user as they're delivered.

The alternative would be the log every single request and then [inaudible 0:27:25.8] them out of some locking system. This allows us to kind of subscribe to events in the system that we really care about for diagnostic purposes primarily today.

Later, we'll add more policy facing future. So for instance this service can't talk to this service or can talk to the service, or I want to allow the SLA between these services to be two 9's or the SLA between these two other services is three 9's. Really kind of being able to set the policy on

the communication between applications rather than on their scheduling parameters with things like that.

[0:27:59.1] JM: Today, the use of Conduit, this new service mesh that you've built, is more about I'm looking at my control plane and I can see the latency between my different services. I can see the number of retries. I can see the number of instances that are being spun up in different contexts. Maybe I can do some stuff around distributed tracing. I guess talk more specifically about what observability features you want from a control plane and how does it fit into the overall — Because like observability is like a big term. Like are we talking about logging? Are we talking about monitoring or metrics? What exactly are we talking about when we say observability from the control plane?

[0:28:44.2] OG: Yeah. I think in some ways we're talking of all the above. We're talking about recording information about the traffic specifically, not about process memories and things like that, but about traffic and making it processsable in some way, invisible by the user. Things we, of course, track are counts of things, so the number of requests, the number of successes, the number of failures. We also talked about latency distributions and size distribution. So how long did it take to get the response to this request?

The way that Conduit track this is in a very kind of detailed way, and so we actually can get this down to the method, the HTTP method and the path and even down to specific headers if [inaudible 0:29:24.1] configure that in the future.

So we can request that fail to the post endpoint. Have this latency distribution where is request that succeed to the put endpoint, have this latency distribution. It really lets you kind of slice and dice the behavior of your application without having very specific code instrumented.

As we start to talk about distributed tracing and things like that, that will of course fall under the proxy over time, but the focus for Conduit right now is to be absolutely zero configuration, zero code change. If we're going to support tracing, that requires some sort of application buy-in to forward headers or some other sorts of information from request to request.

We haven't tried to start there. We want Conduit right now to be — You don't have to configure anything. You just install it and you get visibility for free, and we'll be adding TLS and things like that in the near future as well just so you get these features for free without having to like learn a bunch of new things about the service mesh to take advantage of it.

[0:30:21.5] JM: Am I doing anything today at the control plane to do load-balancing, for example? Because like I think in an ideal world, you might look — Well, correct me if I'm wrong, but you might be looking at your control plane or you might configure something in your control plane where if there is a significant latency between these different communication points, maybe I want to add more networking bandwidth somehow or add another service instance of something. I might want to increase some other configuration thing in response to where I'm bottlenecking, like where the latency is occurring. How am I programmatically doing something that is actionable at that control plane?

[0:31:11.0] OG: Yeah. That's a great question. There's kind of two main strategies for this. One is in a proxy, much like in Linkerd, we'll be doing latency wear load-balancing [inaudible 0:31:19.6] breaking and all of those kind of baseline operational things. So the way that works in Linkerd is that we actually measure the response latencies for every single request to each endpoint and we use that information to inform how to distribute future requests. We basically have a load profile for the entire cluster at each proxy.

As I was saying before, the control plane has this pluggable telemetry system, or it will become more pluggable overtime at least, and that's where we'll be able to integrate these kind of hooks. You can detect these events in the system and say, "Well, my latency is increasing," and then react to those. Even today in Conduit, we install Prometheus with that, and o you can use Prometheus's alert manager and things like that to have triggers based on latency information, for instance, to do auto-scaling or things like that.

We don't want to be too prescriptive on and what those remediations are right now, because I think it requires some knowledge to make that work right, but overtime as we get more insight into what the applications are doing, all those things become possible in the control plane.

[0:32:18.8] JM: Okay. Maybe I'm wrong. On the proxy level, you've got all the sidecar proxies. You got a sidecar proxy that runs with every application pod, and those proxies can say, "Hey, we're getting a lot of latency for this thing or we're having trouble responding quickly. We need to scale up this the proxy." The actual proxy instances can say the service needs to scale up. Is that accurate?

[0:32:44.0] OG: The proxies are kind of dumb. The proxies is really just saying, "Hey, here's latencies," and then the control plane is getting all these information and can look at as an aggregated cluster view of the thing and say, "Oh, look. I've hit my SLO. Here are my objective. Okay, my latency is out of the P99 latency. Maybe I have to scale up more instances [inaudible 0:33:03.8] to satisfy that." But that can't be done in a totally application agnostic way, right? Adding more middle tier services doesn't reduce latency if your database is on fire.

[0:33:13.1] JM: Right. Okay. So the control plane is really where the commands get issued back to the data plane, the different proxies, and it might say to one of the proxies, "Hey, we've detected this kind of issue and you need to scale up to more instances."

[0:33:27.9] OG: Yeah, and the proxies themselves won't be doing the scaling. The control plane may talk to the Kubernetes API to scale out a replica set, for instance.

[0:33:35.8] JM: Oh. Okay.

[0:33:37.1] OG: Yeah. The control plane will be that thing that's talking to other control APIs like Kubernetes and reading telemetry data and then doing something with Kubernetes to an acted change.

[0:33:47.0] JM: Cool. Okay. For somebody who has not actually operated Kubernetes like myself. I'm looking to get on that this Christmas break. That's one of my of my tasks to do in the next week is actually spin up a Kubernetes cluster for the first time.

[0:34:00.1] OG: It's a fun time. You'll enjoy it.

[0:34:01.5] JM: Is it? Okay. Great. I'm looking forward to it. I won't be an armchair architect anymore until then. So let's say I'm getting this information from my proxies to my control plane. The control plane says, "Okay. I want to scale up one of these replica sets," which is like the abstraction of numerous replicas of a service. What happens during that communication between the control plane and the Kubernetes API and what is the command that the control plane is issuing to the Kubernetes API, and how is the Kubernetes API broadcasting back to that replica set, the set of service instances to scale up? Just for people who are a little newer to Kubernetes, how does that process work?

[0:34:43.7] OG: For sure. You might have something that sits in a control plan that's just reading telemetry and looking at different — For four conditions or maybe there's a Prometheus alert manager that's triggering some events based on some SLAs or something like that, and when these events are triggered, you can have a program that's for instance written Go. There is a nice Kubernetes API client and you can talk to the Kubernetes API and say, "Well, this deployment —" So let's say there's a user service. Every service is a user service. We can take a deployment and read it and say, "Oh, there's only five instances of this deployment, and now I'm failing my latency objective. Maybe I need to schedule more."

What we can do is just push a new YAML object into the Kubernetes API and it's literally HTTP put. It's a little easier if you use the go client though, and update the object and you can increase this replica set to 10 or something or it's even just a simple CLI. You can say, "[inaudible 0:35:38.7] scale 10," and then give the deployment name, and that'll tell the Kubernetes API that there need to be more instances of this particular deployment and find places in the cluster to start them.

[0:35:51.1] JM: Got it. And so the future of the control plane is to have some more features around security and other — I guess more —

[0:36:03.4] OG: More control.

[0:36:05.2] JM: More control. Yes, exactly. What kind of security policies would I want to implement at the control plane? What is that actually mean?

[0:36:13.0] OG: There's a kind of variety of policies, and one that we kind of need to start with, there are various products around us right now like Spiffy. It's just having identity on both ends. So if I get a request from another service, how do I know what service that actually came from? That could be a staging service. That could be production service. That could be a service pretending to be another service.

Just being able to establish identity between service is cryptographically, like with [inaudible 0:36:36.9] that says, "Okay. I actually can be assured that the service was run by this other user who is calling me." That's something I think we'll be doing in the short term, is working with these other systems like Spiffy to just integrate with that basic level of identity in the system so that we know where requests are coming from.

[0:36:55.9] JM: That sounds really useful, because this is actually a common occurrence where, "Oops! I accidentally integrated my staging server with production."

[0:37:04.3] OG: Exactly. We have to have that base level of identity in the request streams in order to start to put in the policy of, "Hey, staging can't talk to this production database." But before we have identity, we can't really do properly.

[SPONSOR MESSAGE]

[0:37:25.0] JM: If you enjoy Software Engineering Daily, consider becoming a paid subscriber. Subscribers get access to premium episodes as well as ad free content. The premium episodes will be released roughly once a month and they'll be similar to our recent episode, The Gravity of Kubernetes. If you like that format, subscribe to hear more in the future. We've also taken all 650+ of our episodes. We've copied them and removed the ads, so paid subscribers will not hear advertisements if they listen to the podcast using the Software Engineering Daily app.

To support the show through your subscription, go to softwaredaily.com and click subscribe. Softwaredaily.com is our new platform that the community has been building in the open. We'd love for you to check it out even if you're not subscribing, but if you are subscribing, you can also listen to premium content and the ad free episodes on the website if you're a paid subscriber. So you can use softwaredaily.com for that.

Whether you pay to subscribe or not, just by listening to the show, you are supporting us. You could also tweet about us or write about us on Facebook or something. That's also additional ways to support us. Again, the minimal amount of supporting us by just listening is just fine with us.

Thank you.

[INTERVIEW CONTINUED]

[0:38:59.4] JM: Now that we've given some basic introductions for what the control plane does and what the data plane does. By the way, for people who are still confused, at this point we've done I think four or five shows about service meshes and they should just go back and listen to those other episodes. So let's get back to the wonkiness. I'd love to know more about designing a service mesh, because this is a new kind of application. Are you were sitting down to architect Conduit, you've got this rapidly evolving Kubernetes ecosystem. You've got your experiences with Linkerd to draw on, you've got Istio, which is a very popular service mesh technology as well and there're a lot of innovations happening there. When you're looking out across the landscape and evaluating the knowledge that you've gained over the last couple years, what were some of the design considerations when you were building Conduit?

[0:39:55.9] OG: The biggest one is keep it simple. Absolutely. We started with Linkerd and we realized that a lot of our users get stuck in kind of the richness of the configuration. You can do so many things and it's so configurable, but it's kind of easy to lose sight of the things that you must do, because you get kind of overwhelmed by all of the things you could do.

We see with people very over complex service meshes, and that's a big concern for me, that the technology, the service mesh technology may not get adoption, because it's seen as this complex unnecessary set of solutions.

With Conduit, we really wanted to focus on the minimal set of things that's not overwhelming to get your head around and that like it's useful immediately and it's not something I have to totally

get everyone on board to get that value out of — That I can do better debugging at least immediately without changing my application at all, without having to learn anything new.

That's kind of where we wanted to start with Conduit, is really just lightweight in terms of resources. It's very kind of, I think, under Megan, the demo I did at Cube-Con is really a tiny proxy, but also lightweight conceptually.

[0:41:00.4] JM: The language of choice for this lightweightness is Rust in terms of the data plane at least. The sidecars in the data plane are in Rust. So as I understand, this is also for memory safety issues. Rust has safety as well as the high speed. Explain more about why rust is the right choice for a sidecar.

[0:41:23.5] OG: Yes. Many of us have been in production operations for a while and watching over the last few years, heart bleed and all of these issues creep up in core libraries that the whole internet depends on. It was a little scary for us. As we started to think about our kind of place in the infrastructure we are, that we'd be linking against these unsafe libraries. It was kind of scary in terms of offering support to banks and those sorts of things, which of course point would like to do eventually.

We certainly really seriously think about Rust and what advantages there would be there. So we started looking at products like Rust-tls. Rust TLS is a reimplementation of TLS that doesn't link against open SSL or any of the C libraries. It reuses a bunch of the assembly code from those, which are been verified and validated externally, but this actually gives us a much kind of better guarantee around at least free after use and all of the big traps that catch open SSL in every C library out there.

Our goal is to have a very minimal C dependency footprint and really have tried leverage native safety in Rust for a lot of security and correctness guarantees. It's not just security, but I think it will allow us to write more correct code if we can rely on the language features properly.

[0:42:42.7] JM: When you talk about memory safety, you're not talking about garbage collection. You're talking about heart bleed. That was like buffer overrun error, right? Like there

were some problem where if you requested things correctly over SS — How do that bug work again?

[0:43:02.2] OG: You could send something to the server that would make it read from improperly freed memory that wasn't cleared and you could start to read out session keys and things out of arbitrary memory. That's just because the C compilers don't enforce anything around your access to buffers. So Rust a lot of. It has no GC. Not a garbage collector language. It's all done at compile time. All of the reference counting an all of these things are just managed by the compiler for you. Makes compilation a little slower, which is hopefully gets improved over time, but we get a lot more out of that compilation than we do on the Go side, and the control is written in Go so that we want a lot of people to contribute to the controller than the data plane, which shouldn't hopefully be boring code that doesn't change all that often once we get all the features we need into it.

[0:43:48.2] JM: So were there some issues in the Linkerd sidecar that relate to safety issue?

[0:43:56.0] OG: Well, so Linkerd is super pluggable. Like I said before, you can write these plug-ins for any services covered back in all sorts of different policies plugins, etc. But that means that people are writing their encoding putting in the data plane. More generally, the data plane does a lot —Linkerd can do a lot of work there. We didn't have any big security incident that triggered this, it did make us, from a support point of view, think that it was a lot harder for us to guarantee correct behavior at all in the data plane if other people are adding code in the data plane directly. What we wanted to do with Conduit is have a very configurable pluggable control plane that works via an API with the proxy, but the proxy is basically bolted down and doesn't have a lot of functionality built into it that isn't explicitly expressed via the API.

[0:44:46.5] JM: Did you have to make any opinionated decisions? Because if you're saying that there were people that wanted to plug-and-play and do all kinds of crazy stuff with their Linkerd sidecar proxy, then probably if you wanted to tamp that down building Conduit, you had to make some constraints.

[0:45:06.0] OG: Oh, yeah. Linkerd, you can build many different kinds of service meshes with Linkerd and people — That's why they all look so different. Conduit will be a much more

opinionated limited system, and so we've started with a very constraints of the features not because we don't know how this future should work, but because we just want to start with the essentials and not overwhelm people or add a lot of complexity in the system early on. So we'll be layering these new features, but it'll be, I think, in a much more opinionated targeted constraint way.

[0:45:36.7] JM: You said the communication between the data plane service proxies and the control plane, the service mesh that's over GRPC?

[0:45:46.1] OG: Yeah. That's over GRPC and all of the interfaces within the control plane are also GRPC, and so we kind of have a small micro-service in the control plane.

[0:45:54.2] JM: Explain what GRPC is.

[0:45:56.1] OG: GRPC is a technology built like Google and some other folks that kind of is the newest version of an older technology of thrift or protobuf. So what those are, are ways to describe APIs via a small domain specific language, DSL. So I can just say the user service has a get user method and a delete user method, and a user struct has an age field and a hair color field or whatever it is. Just the way they kind of uniformly describe structures and service interfaces independently of a language implementation, and then we can use the [inaudible 0:46:40.4] tool to generate GRPC bindings in any number of languages, a PHP or node or are Rust. Now we wrote a Rust GRPC generator, and I've written a Scala one in the past.

It is a way for us to generate services programmatically, and then they all communicate over HTTP/2. Conduit right now actually only supports HTTP/2. We'll be adding HTTP/1 support in the near future. Yeah, we've been really kind of focused on that forward looking set of primitives and technologies there.

Just for some historical context. So I think if you want — Many of your services throughout the internet probably communicate with HTTP requests, get put restful requests, but they're sending JSOn as the content or they're sending XML as the content. You could just as easily send GRPC or protobuf's as the content. Is that correct?

[0:47:33.8] OG: Yeah. GRPC can even generate JSON body just a way for building the clients and server code that generate that. So when I write a GRPC server, I don't even know whether it's JSON or protobuf on the wire. The library can choose that for me. I just know that I returned an object of this shape and it's somehow going to get serialized on the wire for the client.

[0:47:55.9] JM: Did GRPC — It was not a new serialization protocol too?

[0:47:59.6] OG: No. GRPC itself can use Thrift or JSON or a protobuf. Protobuf by default, but it can actually have other serialization formats as well.

[0:48:07.9] JM: Okay. Interesting. I didn't show pretty recently about protobufs and this other serialization protocol that this guy Kenton made called Cap'n Proto. Have you heard of that one?

[0:48:20.8] OG: Oh, yeah. Of course. Yeah.

[0:48:22.9] JM: Yeah. Why hasn't the world updated to that one? It sounded like it was a little more speedy?

[0:48:27.6] OG: Because it's not all that speedy. These things exist in organizations where they have a lot of existing protobuf or a lot of existing thrift. Twitter I think will never move away from Thrift just because they've got so much investment there, and I'm sure Google will never move away from protobuf for similar reasons. Once these technologies get entrenched in organizations, it's just hard to rip them out.

[0:48:48.0] JM: Yeah. I guess you got to be a thousand X better, 10 X better something to really get people to do that kind of thing.

[0:48:54.0] OG: Yeah. The CPU cost is not the only cost.

[0:48:56.8] JM: Yes. Okay. Let's shed a little more light on this communication. Like when am I writing that GRPC plug-in or a service and what does that look like?

[0:49:07.4] OG: Yes. Let me kind of walk-through maybe a life for request a little bit. When your service tries the sender request, it'll just say, “Okay. I want to call the user service and send this request there.” When that goes out, when you injected the sidecar into your application, we add our IP cables reroute rule that sends all that traffic through the proxy. So you don't have to change your application to send traffic to the proxy. It all happens automatically.

When the proxy gets a request, it's going to look at the header. It’s going to look at the host header and say, “Where is this request going?” If it knows, if it's looked up a request like this before, it will just forward it along over that load balancer to that endpoint, otherwise it'll send the GRPC request to the controller and say, “Hey I need a look up for the user service. Give me back the set of IP addresses that make up the user service. Actually, IP import address.” Then that’s over GRPC request and the server responds with just the stream of updates watching the Kubernetes API basically for services career changes.

Then once the proxy gets all of those addresses at start, it builds connections up and starts to send or dispatch the requests to whichever endpoints are available that I can send them to. So each request to it gets, it may send to a different endpoint in that cluster according to whatever load-balancing logic applies.

Then as these requests go through the proxy, the proxies also counting them and measuring the latencies and looking at the status codes and things like that and building up a collection of data over some time window. By default, every 10 seconds, we flush out a bunch of data to the control plane of recent telemetry. Yeah, we keep buffering up data and then just flush it out to the control plane, which ends up in Prometheus and then is queryable in the UI or via the CLI.

[0:50:55.3] JM: If I’m an operator, or I guess a developer. Who even knows what the differences anymore? What's my interaction with the service mesh technology on a day-to-day basis or when am I writing these plug-ins or this code for service mesh?

[0:51:10.3] OG: Yes. The plug-ins are going to become when you have an automation. Like I said, we have Prometheus in there by default. We have a telemetry API. So you don't actually have to write a plug-in into the mesh to start to read data out of it or things like that. But let's say you wanted to plug in a different — Maybe you want to send some of this data to Datadog or to

your other in-house telemetry backend. Well, you then might write a plugin that's going to fill through the data according to what you actually want and send out some report to another service, or let's say that you want to have slightly different services discovery logic, and the way that you look up services isn't the default way that Kubernetes looks up services for instance, because you have one of these hybrid environments because you're migrating to Kubernetes still. Then you write a destination plugin that we call the destination services, the service discovery service. You might write a destination plugin that applies your service discovery logic before falling back to our default service discovery logic.

It's going to be for kind of overriding behavior. Later, this will also come into policy when we start to say rate limiting policy or can this service talk that service? We'll be integrating with things like OPA, the open policy agent APIs that you can plugin to the controller. Probably also the XTS APIs. These are set of APIs that Istio uses with Envoy. I expect you'll be able to plug in those APIs as well into the control plane overtime.

[0:52:35.4] JM: Okay. Yeah, that was my next question was, you look back two and a half years and you see how Kubernetes and Masos and Swarm and the difference between them led to a slower pace of innovation than we have today. Today, at Cube-Con, it was just insane about you could tell how fast everything was moving. It was basically because people were saying, "Okay. We're using Kubernetes. Everybody's using Kubernetes. Get on the boat or get off of it."

Because of that, people can make certain assumptions. Open-source project can make assumptions and enterprises can make assumptions and we're all growing in the same direction and it's great and it makes me wonder, is the service mesh another winner-take-most market or is it — Could you have some kind of standardization like we're seeing with the open container initiative, kind of a container definition that is more broad. I think we're also seeing this with serverless. I think the serverless people were meeting and having their — I don't know. Some back office discussions at Cube-Con. Like how are we going to standardized serverless.

Yeah. I mean, if you can standardize it, it almost doesn't matter if you're running Istio or you're running LInkerd or you're running Conduit, because you've got a standardized plug-in interface.

[0:53:56.3] OG: Yeah. I think aspects of those things will standardize, like service discovery, a natural thing for an API to become boring. But I don't expect all of service meshes to become boring. I think they'll be parts of it that become boring and parts of it where it's really worth differentiating in terms of features or opinions. We'll have to see what the boring parts end up being, versus the really interesting features where it's going to be helpful for there to be multiple ideas around how X, Y, or Z should work. It's not an all or nothing thing in my mind. It's definitely pieces of it.

[0:54:27.0] JM: Yeah. Okay, that makes sense, because you look outt, for the last probably 10 or 20 years, in an expo hall at a conference and you could go and there's five or six or 20 different logging and monitoring providers and all them have great businesses. All of them have popular platforms. It was not a winner-take-all in logging and monitoring. You could see service mesh be the same thing.

[0:54:51.1] OG: Yeah, and all of these logging things have standard features and they differentiate in other ways. I think service mesh will be the same way. Buoyant, obviously our mission here is to provide that immediate value and work with the people we've been working with for the last two years to solve real problems today. But as we see the industry really adopts certain things, like Kubernetes, we've standardized around that. We're going to build into a bunch of their APIs. As our user bases are really clamoring for certain features and standards, there's no reason Buoyant won't adopt those.

[0:55:24.7] JM: I know we're almost out of time, but I had a conversation with Brendan Burns recently on the show and it kind of blew my mind. He said something that really blew my mind where he was talking about where Kubernetes is going and what the implications are. The thing that really stood out to me was he was talking about how you could have a new software purchasing model where you buy basically a binary and you just run it on any Kubernetes. This could have business implications where you could now buy a Zendesk binary and you could have Zendesk on your Kubernetes, and there you go, you've got Zendesk and you no longer need to pay Zendesk. You just pay your cloud service provider for server runtime, which is going to be much cheaper than paying the SaaS abstraction that is just hosting that for you. I was like, "That's kind of cool. Like maybe that'll happen, maybe it won't, but it's at least a cool idea," and the service mesh seems like another layer where this is something that's kind of new and there

seems like there's probably some crazy business opportunities that can be built in the upmarket like 5 to 10 years, or maybe three years or two years. I don't know. What happens? What's the upmarket effect of the service mesh?

[0:56:39.1] OG: We're not working on service mesh because we want to have support contracts for service mesh users. We're working on service mesh because it will think unlock, like you say, a lot of business potential. I think the micro-services have the opportunity to really change the way that teams and organizations work, engineering teams and organizations, because people have more autonomy and individual responsibility. I don't know. It was a really personally changing experience to see Twitter and all of the opportunities for individual leadership that kind of come out of that sort of organization operationally.

I think that that's where I see companies like Buoyant and other companies in the industry really playing, is how do they changed businesses to be better using these technologies? The technologies are kind of — These are things that have to work, have to be open-source, have to be available and making them work right is not an interesting business, but using these things to make businesses work better is very exciting on my mind.

[0:57:38.3] JM: Okay. Well, it's a bright future. Oliver, thanks for coming on Software Engineering Daily. It's been great talking to you, and give my best to William. I always love talking to him. He's a hilarious guy.

[0:57:47.0] OG: Thanks for letting me nerd out about this topic. It's really fun for me. Hopefully audience enjoys it.

[0:57:51.9] JM: Oh! Of course. I'm sure will be doing it again in the future.

[0:57:54.8] OG: All right. Great, Jeff. Thank you.

[0:57:56.3] JM: Okay. All right. Thanks, Oliver.

[END OF INTERVIEW]

[0:58:01.1] JM: GoCD is an open source continuous delivery server built by ThoughtWorks. GoCD provides continuous delivery out of the box with its built-in pipelines, advanced traceability and value stream visualization. With GoCD you can easily model, orchestrate and visualize complex workflows from end-to-end. GoCD supports modern infrastructure with elastic, on-demand agents and cloud deployments. The plugin ecosystem ensures that GoCD will work well within your own unique environment.

To learn more about GoCD, visit gocd.org/sedaily. That's gocd.org/sedaily. It's free to use and there's professional support and enterprise add-ons that are available from ThoughtWorks. You can find it at gocd.org/sedaily.

If you want to hear more about GoCD and the other projects that ThoughtWorks is working on, listen back to our old episodes with the ThoughtWorks team who have built the product. You can search for ThoughtWorks on Software Engineering Daily.

Thanks to ThoughtWorks for continuing to sponsor Software Engineering Daily and for building GoCD.

[END]