EPISODE 811

[INTRODUCTION]

[00:00:00] JM: Consul is a tool from HashiCorp that allows users to store and retrieve information from a highly available key value data store. Consul is used for storage of critical cluster information, such as service IP locations and configuration data.

A service interacts with Consul via a daemon process on the node of that service. The daemon process periodically serves information to the consul server over a gossip UDP protocol, and the daemon can also share data on a more immediate basis using TCP. Consul's functionality has increased recently to add secure service connectivity. Consul connect allows services to establish mutual TLS encryption with each other.

The addition of mutual TLS to the consul feature set is closely incidental with Consul gaining a title of service mesh. Service mesh is an increasingly popular pattern that can encompass a variety of features; load balancing, security policy management, service discovery, routing and more. Tools which offer self-described service mesh functionality include Linkerd, Kong, AWS App Mesh, solo.io, Gloo and Google's Istio open source project.

Paul Banks is the engineering lead of Consul at HashiCorp. He joins the show to talk about the service mesh category and the past, present and future of Consul.

Some recent updates in Software Engineering Daily land, the FindCollabs $5,000 hackathon ends this Saturday, April 15th, 2019. There's still plenty of time to enter your project. Whether that project is open source, it's close source, it's music, it's art, it's wireframes. There are plenty of opportunities to enter projects, and definitely there's time to hack on something and make enough progress to win that first place $4,000 prize. You can find all those details in this episode show notes.

Also, the new version of Software Daily, which is our app and ad free subscription service, is on softwaredaily.com. It's got a lot of clean up. We are finally working on cleaning up the apps. I know the apps have their performance issues and I am so grateful that people have tolerated those performance issues and continued to use the apps. We really appreciate that, because we do have a vision for where these things are going. We would eventually like them to play audio properly. I realized that it's not happening right now. I mean, the audio does play, but it's certainly not as good of an experience as your podcast playing app.

So if you have performance issues with the app, but you see the vision for the app, we would love to get your reporting of those bugs. You can go to our FindCollabs, the Software Daily FindCollabs, which is also in this episode's show notes, and there's also some positions there. We're looking for help on Software Daily, our open source project. We need help with the Android engineering, AQ, machine learning and more, and we are infesting right now over the summer into improving the open source experience. I know that has been deficient in other ways as well. So we'd love your feedback in the FindCollabs.

With that, let's get on to today's episode.

[SPONSOR MESSAGE]

[00:03:38] JM: Deploying to the cloud should be simple. You shouldn't feel locked-in and your cloud provider should offer you customer support 24 hours a day, seven days a week, because might be up in the middle of the night trying to figure out why your application is having errors, and your cloud provider's support team should be there to help you.

Linode is a simple, efficient cloud provider with excellent customer support, and today you can get $20 in free credit by going to linode.com/sedaily and signing up with code SEDaily 2019. Linode has been offering hosting for 16 years, and the roots of the company are in its name. Linode gives you Linux nodes at an affordable price with security, high-availability and customer service.

You can get $20 in free credit by going to linode.com/sedaily, signing up with code SEDaily 2019, and get your application deployed to Linode. Linode makes it easy to deploy and scale those applications with high uptime. You've got features like backups and node balancers to give you additional tooling when you need it. Of course, you can get free credits of $20 by going to linode.com/sedaily and entering code SEDaily 2019.

Thanks for supporting Software Engineering Daily, and thanks to Linode.

[INTERVIEW CONTINUED]

[00:05:15] JM: Paul Banks, you are an engineering lead at HashiCorp. You work on Consul. Welcome to Software Engineering Daily.

[00:05:21] PB: Thank you. It's great to be here.

[00:05:23] JM: I'd like to start by discussing the category of technologies that are called service meshes with you. Why is there a need for this abstraction called service mesh?

[00:05:34] PB: That's a great question, and I think that the answer comes down to a couple, as there's kind of two ways to look at it. So the answer from a kind of a business perspective is that businesses are moving from designing software and writing software in a very kind of monolithic fashion, where they have this one big application that get shipped to all their servers and kind of does everything.

Moving towards – The hot term is microservices, but service oriented architecture. Kind of however you want to break that down. One of the big – There are a lot of benefits. We won't kind of go too much into that. There are lots of benefits to this in terms of scaling your teams and being able to deploy independently kind of separate your failure domain, scale independently, these sorts of things.

But one of the big problems with it is kind of operational maturity, and you now have lots more network calls. It's much harder to debug things that are running on different machines talking over the network. All of these services that need to talk to other things kind of have to start reinventing some of the patterns for reliable communication. Things like handling errors correctly. Backing off when you hit errors. Making retries to a different instance of the service. Discovering even where all of your upstream dependencies are. All of these kind of problems that you didn't have when you had a monolithic application. So service mesh from a business value point of view comes from trying to solve this for this new kind of paradigm where things are very separate services.

[00:07:07] JM: The service mesh architecture consists of a data plane and a control plane. Describe those two parts.

[00:07:14] PB: Sure. So as with most kind of systems, infrastructure systems, data plane and control plane are fairly common terms. So the control plane is really the source of truth for where things are running, kind of a registry of services, if you will. It's responsible for configuring the data plane. So configuring where to connect to things, what sort of settings for things like retries on errors and timeouts and similar, also TLS certificates and security between different services.

The actual application traffic itself though just flows through the data plane. So this is what's transporting HTTP requests or TCP packets in between the different service instances. In the case of service mesh, specifically, the data plane consists of sidecar proxies, which are generally proxying at layer 7. So usually HTTP or GRPC or something like that.

They can tend to be able to do layer 4, so just plane TCP for things like database protocols. But most of the value of kind of the current service mesh market is coming in in kind of layer 7 or where proxying where, you know, it knows whether the HTTP request was an error or not and whether it can retry it and so on.

[00:08:32] JM: Service mesh is often described as something that's used to do so many different things; load balancing, service discovery, security policy management. There's a big bundle of things that often gets lumped in this service mesh category. Why is this single abstraction used for so many different things?

[00:08:52] PB: I think that's a great question, and I think it comes to kind of the second view point that I was talking to earlier about kind of what service mesh is, and that's kind of trying to understand where it came from. Service mesh, as a term, was kind of first coined I think by Linkerd, which is a project that came out of Twitter, and it came out of their experience.

So at Twitter – I forget, like 5, probably more years ago, they were working out how to kind of write these robust services and enable different teams to write services without resolving all of these problems. So they wrote this library called Finagle, which is a library for the JVM, which you basically – It's a framework for building services, and it solves a lot of the kind of plumbing problems for you.

So it has different mechanism to do service discovery from ZooKeeper and different sources of registration. Then it layers up like the RPC protocol and kind of layer 7 aware things, smart load balancing, circuit breaking on errors. These kinds of things that all their teams were sort of having to solve individually. They kind of built that into a framework.

So that worked very well for them. So like the abstraction was what are all the things that are plumbing that are not like – We don't want every different service team to go figure out their own way to do it. We just want one good way of doing this, and then we can just ship that. So that's kind of the abstraction.

The problem that they found with that was because it's written as a library, it's compiled into every single service. So when you want their support in your protocol, you're back to the problem that you thought you'd solved with microservices, which is that you have to go get all of your different service teams to recompile with the new version of your library and redeploy and sync and kind of roll out this new version so that you can start using these new kind of features.

So that was a really big motivation for kind of taking that abstraction and moving it out of the application process itself, and this is kind of where the idea of a sidecar proxy really comes into its own. Because now the application itself, the deployable unit that your service team writes doesn't have to worry about any of that stuff and it doesn't have to worry about getting redeployed or recompiling itself if you ever want to change any of that plumbing.

So if I recall correctly, I think it was some people from Twitter who'd kind of worked on this problem who kind of came up with Linkerd, which very much was just the finable framework, but compiled into a separate proxy that sat next to your application instead of just being a framework. I guess the whole kind of space has evolved from there. We have Envoy coming along as a Lyft's version of the proxy part, and then they have their own control plane. Then kind of a whole set of new products coming out based on that proxy and others.

[00:11:56] JM: Yeah. So there's a lot of different ways that this service mesh can be implemented. Can you describe some of the different architectural patterns that we've seen? I mean, you described one with the Linkerd – Well, I guess with the initial Finagle abstraction where you have this library. But what are the other architectural patterns for having this service mesh?

[00:12:18] PB: I think the service mesh is one of these new terms that everyone has a slightly different definition of. So just for one of trying to keep it as a relatively firm thing. For a lot of people, the idea of service mesh is actually very tightly coupled to this idea of a sidecar proxy in the data plane. While for others, they talk about service mesh, but they really just mean like anything that helps you kind of plum things together.

So in the broader scope of like how do you solve this problem of connectivity between your services, then yeah, one solution is you have like a baked in hybrid that can do that for you that's kind of right there on your application. Kind of Finagle or Thrift style. But I'd say most of the market is kind of using the term service mesh very specifically to mean deploying a sidecar proxy next to your application and then having a control plane to configure those proxies to talk to each other without the application necessarily having to know to some extent or like having to really like re-implement any of that functionality.

[00:13:22] JM: Big companies have had this service mesh thing for a long time. They may not have called it service mesh, but Amazon has one. Uber has one. Lyft has one. They fulfill similar goals, sometimes overlapping, sometimes somewhat disjoint. How does the design of service meshes vary across some of these big company architectures? Have you talked to many of these big companies?

[00:13:50] PB: I've not directly. I'm kind of familiar with the way Lyft have set out – Obviously, Lyft wrote Envoy, which is a very common proxy and it's used at Consul Connect. It uses Envoy as one of the proxy options there. Istio is based on Envoy. Amazon just released their App Mesh offering, which is based on Envoy as a proxy in their control plane. So it certainly has a lot of attention now. That came out of Lyft's internal service mesh. So they kind of have their own internal control plane. They call it management server, that configures all of these Envoy instances. Yeah, I'm not super familiar with kind of internals at Amazon or any of the others.

[00:14:29] JM: No problem. As we get into HashiCorp and Consul specifically, Consul has existed for some time, at least five years, maybe longer. Consul was there before the term service mesh was in the public Lexicon. Explain what role Consul has historically fulfilled for people.

[00:14:51] PB: Yeah. That's a great question. So Consul started off as kind of focused on being a service discovery tool. So your workloads would register themselves with their local agent. Then because there's a local agent on every host in your data center, it kind of solves this problem of needing to work out who to talk to first. So you have a local agent. You can just talk to a local host and then that agent can tell you where everything else is running.

So it can do that through just a plane DNS interface. So you can just do DNS queries and get back IP addresses. Obviously, we all know the dynamic port problem with DNS, and you can use SRV records, but then nothing really does these SRV records. So you can choose to meet that conundrum however you will. But then there's also this HTTP interface, which allows you to kind of block, watch for changes in the cluster. The way people have typically used that is using some extra tooling, like we have a tool called Consul Template, which allows you to kind of watch for changes in the services instances in your cluster and re-render a config file on disk and optionally run a command like to [inaudible 00:16:02] a proxy or something.

So people have used this kind of setup for years to kind of dynamically configure their load balancers and their firewalls and whatnot so they can kind of have Consul as this source of truth for where everything is and have everything dynamically pick up the changes in a network.

[00:16:22] JM: Describe the software architecture of Consul in more detail.

[00:16:26] PB: Yeah. So I mentioned there that Consul has a local agent that runs on every host in the data center. This agent if fairly lightweight, and all the agents join together using a gossip protocol, so a decentralized membership protocol that scales very well to very large clusters. We've successfully run it on clusters of more than 10,000 machines sometimes, several tens of thousands of machines in a single gossip pool.

The purpose of this gossip layer is it really provides two benefits. The first one is discover of where everyone else is, which is how we can bootstrap this service discover thing, because every agent is able to discover all the other nodes. It's then able to discover where the central servers are. I'll mention those now.

So as well as an agent on every host, within each data center, you set up usually three or five machines as Consul servers. You use three or five, because they replicate what we call the catalog, which is the central state of where everything is in the cluster. They use the Raft Consensus Protocol to make sure that's strongly consistently held in the service, and because they also gossip with everything.

So when the local agent comes up, it's able to gossip with its peers and discover where the servers are. From then on, it can redirect any DNS queries to those servers. So you can get strongly consistent results about where things are running in your cluster at any point without you having to even know what the DNS name of your service is, because you just talked with your local host.

[00:18:09] JM: So you describe this agent, daemon. Does this agent run on the service node itself or does it run in a sidecar container?

[00:18:20] PB: So in a kind of traditional bare metal or VM setup, which is where Consul kind of was originally conceived, this agent runs per physical host or per VM. The idea with that is the second part of that gossip protocol's purpose. I mentioned discovery is the first one. The second part is it's a distributed failure detector. Service gossip protocol can very quickly and reliably detect if a node is failing and if gossip messages aren't reaching it and it can probe that from several other nodes to get a strong suspicion that it's failed. It can then propagate that information very quickly throughout the whole cluster.

So the idea is that you run the agent on the node and then Consul can use that membership information, that failure detection information, to automatically remove that node services from the catalog as soon as it sees that it's failing. So DNS records will stop returning that node's IP address, and if you're HTTP API or start to see its health check failing and you can remove it from your load balancer and so on.

[SPONSOR MESSAGE]

[00:19:30] JM: I have been going to the O'Reilly Software Conferences for the last four years ever since I started Software Engineering Daily. O'Reilly Conferences are always a great way to learn about new technologies, to network with people, to eat some great food, and the 2019 O'Reilly Software Architecture Conference is for anyone interested in designing and engineering software more intelligently. There are actually three software architecture conferences. One is in New York, February 3rd through 6th. In San Jose, June 10th through 13th; and in Berlin, November 4th through 7th.

So if you're in New York, or San Jose, or Berlin, or you're interested in traveling to one of those places, take a look at software architecture. You can get a 20% discount on your ticket by going to softwarearchitecturecon.com/sedaily and entering discount code SE20.

Software architecture is a great place to learn about microservices, domain-driven design, software frameworks and management. There are lots of great networking opportunities to get better at your current job, or to meet people, or to find a new job altogether. I have met great people at every O'Reilly software conference I have gone to, because people who love software flock to the O'Reilly Conferences. It's just a great way to have conversations about software.

You can go to softwarearchitecturecon.com/sedaily and find out more about the software architecture conference. That's a long URL, so you can also go to the website for this podcast and find that URL. The software architecture conference is highly educational and your company that you work for will probably pay for it, but if they don't, you can get 20% off by going to softwarearchitecturecon.com/sedaily and use promo code SE20.

Thanks to O'Reilly for supporting us since the beginning of Software Engineering Daily with passes to your conferences, with media exposure to different guests that we had early on.

Thanks for producing so much great material about software engineering and for being a great partner with us as we have grown.

Thanks to O'Reilly.

[INTERVIEW CONTINUED]

[00:22:13] JM: The Consul cluster model includes nodes that are running both in client mode and nodes that are running in server mode. Both of these types of nodes are running this daemon, but the interaction between the client and server is of course different depending on which is the client and which is the server. Describe the client server relationship in more detail.

[00:22:37] PB: Yeah. So we often call the agents that run on every machine the Consul client agents, and these are the ones that participate in gossip with other nodes to form this failure detector. But other than that, they mostly just accept local registrations and maintain the truth – The catalog centrally for their own registrations. They're the source of truth for what's registered on them. They do this through RPC calls to the servers.

So if you look at our API, we have both the DNS and the HTTP API, which is served from the local agent. But the majority of both of those requests are actually proxied through this kind of internal RPC protocol between the client and the servers. That's kind of a custom RPC protocol that's multiplexed over just a single TCP connection. So each client makes just one TCP connection up to the servers and then can multiplex all the communication it needs to do to the servers. Then it exposes the HTTP API just on the very edge.

And the servers on the other hand run the middle. They expose this RPC API to clients and then they also participate between themselves in the Raft Protocol as well to kind of make sure all the catalogue, the key value store state and various other features that we have is kind of strongly consistent and replicated between them.

[00:24:02] JM: One of the typical use cases for Consul is service discovery. Explain how service discovery would work in Consul.

[00:24:12] PB: Great question. Similarly to kind of how I've described, the idea is that there's a local client running on the machine. So as you're deploying your service either through a file on the disk or through an API call just to local host, you register your service instance and you say, "I have an instance of service web and I'm running on port 8080."

Then that agent will then make sure that that service is registered in this catalogue on the service, and it performs a periodic NT entropy to make sure that the state is consistently seen and consistently correct on the servers. Then any other node that may want to discover whether web instances are can either issue a DNS query for web.service.consul to its local agent, which typically use setup using your resolver config on the host server or DNS requests for .consuldomain go to the local agent. The local agent then resolves that internally through RPC and finds all of the IP addresses of all of the nodes that have a web instance on them and then it returns that, and the local agent actually makes the DNS A Record response back to you.

So that's the simplest way you can discover where these services are running through DNS, or you can use the HTTP API. If you want to write an application that's kind of aware of Consul and can do more sophisticated load balancing choices or discover choices, then it can directly interact with our API, or you can use one of these tools, like Consul Template or [inaudible 00:25:50] Consul, which will use our API and then render our the results to, say, file or to environment variables and keep those up-to-date as they change.

[00:26:00] JM: Service discovery is one nice tool that we can use with Consul. What are some other patterns for how people use Consul? Maybe you could give an example of how another one works and the engineering behind it.

[00:26:14] PB: So Consul has kind of – As with most things that have been around a little while has grown a lot of different features that are either sort of add-ons to that main service discover use case or kind of related. So I think the biggest kind of different features we have is we also have a built-in KV store, which is explicitly not meant to be used as a replacement of your database. If you're looking for a KV store for your next service or database, it's probably not the right choice, and the same way that ZooKeeper is probably not the right choice.

But it is great as a metadata and as a configuration store, because it's built on top of that Raft Consensus, so it's strongly consistent and because you can watch for changes in it. Then on top of that KV Store, there's a mechanism called sessions and lock. As well as just storing KVs, you can actually register sessions which get attached to the surf health checks or any other health checks you want to register on the local agent, and you can then take a lock on a particular key in the KV store, and you can use this to implement distributed locks between different processes in your service.

If one of those processes goes away and its agent health check start to fail, then the KV store can automatically release that lock for you and one of your other instances will see that and will claim the lock. So it can help with coordination problems like that.

I realized I neglected to mention there that as well as this kind of surf gossip-based health checking of all the nodes, when you register your service, you can also register custom health checks with that. So you can have the local agent perform a variety of different source of health checks against the application, which can be anything from running a script and getting kind of [inaudible 00:28:06] style, like critical warning, "Okay. Status from a script." You can have it call an HTTPM point in your service. You can have it just try and open a TCP connection. There's a whole bunch of different checks that the local agent can perform. If any one of those fails, then that gets propagated through the network and your instance is sort of taken out of the server discovery results.

[00:28:29] JM: It's worth talking a little bit about the different ways that nodes can communicate with Consul and with each other, of course. So there's gossip. There's the gossip protocol, and that's going to be a lazier, lower bandwidth communication protocol. Then there're also reliable TCP calls that can be made in certain context. Can you explain the network communication across a Consul cluster? When are we doing this lazier gossip protocol and when are we doing the more reliable TCP calling?

[00:29:06] PB: Yeah, that's a great question actually, and it's even more subtle than your question in place, in fact, because the gossip layer uses both TCP and UDP for different purposes. It uses TCP for exchanging the full state. So when you first join the gossip cluster, the first thing you have to do is learn about the entire state of all the members in the cluster from someone else. So that exchange is done over TCP. We also use TCP as a full back for the actual gossip messages not really for any reason other than people forget to open UDP ports on their servers often. So it's just nice. If your gossip is failing over UDP, we actually try it over TCP and then just throw a warning in your logs that you've probably not configured the network correctly.

The actual kind of regular gossip messages, if you're familiar with gossip, is based on the SWIM protocol with some extensions. Very, very high level, each node every second picks a random other node in the cluster and just sends a ping message to it and expects to get an act back in the small amount of time. If it does, all is well, and it just continuous on.

The fact that it's just one request every second for every node is what keeps the overall bandwidth low. But because of the randomized nature, that's what gives you the kind of quick convergence of like detecting things are down, because with high probability, everything is going to get probed by at least something else every second.

So those messages are UDP. They're not reliable. There are a couple of other features of Consul that are based on the gossip. So there's a feature called user events, where you can send kind of an arbitrary payload as long as it fits in – As long as it's less than about a kilobyte. It has to fit in a single UDP packet with some overhead.

That will be distributed through the cluster through the same gossip mechanism that's doing this health checking, which makes it kind of very efficient to disseminate quickly without actually physically go and contacting everyone. But it's one a best effort basis only. If you do that on two different nodes, there's no guarantee of ordering or that anyone will get it. So it's quite hard to reason about building applications upon that kind of construct, and that was the original reason for adding the kind of central consistent states into Consul to make reasoning about those things simpler.

So that's kind of the only place we use this eventually consistent kind of very scalable gossip system, is for those problems. All of the communication between the client agents and the servers is over TCP and it's just an RPC protocol, so request response.

[00:32:04] JM: People might be thinking at this point, “Okay. This conversation has started off talking about service mesh, and now they’re talking about this key value distributed storage thing.” What’s the connection between these two subjects that we’ve been discussing?

[00:32:22] PB: The key piece to understand that is going back to your question earlier about like talking about a control plane and a data plane. So we’ve talked in separate pieces of Consul’s architecture. But if you kind of take a step back and look at the things we’ve covered, really, the way you can sum up what Consul does overall these features is it is a control plane for your data center. It supports scaling to a large number of nodes, distributing health checks for things across that infrastructure, and efficiently being able to update a central registry of where everything is and what state it’s in.

It also supports arbitrary centralized configuration in the KV store, and it supports an efficient way to distribute updates to both what’s running where and what state it’s in and what should be configured? How should things be behaving? All those things to be centrally configured and then distributed efficiently out to the edges out to each node where they’re needed. So what you have and what Consul has been up until last year is a really great system to build a control plane on.

So as people start kind of moving into this service mesh area and rather than just using Consul to configure that, like edge routers and HAProxies, kind of sitting at the edge of their clusters. The real question is like how does Consul do a really great job of supporting this service mesh use case as well? Almost everything you need is there already.

The biggest thing that we added on top of this was a really strong sense of security and identity for your services. This is actually – This is something that not all service mesh offerings start with. Linkerd really started with communication reliability, L7 things, and it’s only experimentally thinking about configuring mutual TLS between services.

Consul Connect actually came from kind of the other direction, and we started thinking about how do we secure traffic between all of these different services in a way that kind of scales and will work with dynamic workloads, because configuring files is just not something that works in this world kind of scalably .

So we have a big emphasis on setting up mutual TLS. We have a built in certificate authority. Every workload gets unique certificate, and then you can manage your central rule's intentions for which services can speak to which services in Consul, and we enforce that based on these cryptographic identities rather than on IP addresses and so on.

The big benefit of this of a firewall management is it's scale independent. So if you have 100 different web instances and you need to authorize them to access the DB, you don't need to go and make 100 firewall rules in the database, IP tables, whatever. Every time a new web instance comes and goes, you don't need to change the IP addresses that are allowed to access the database, because each one of those instances has the same identity of its service, which is web. That we can just enforce that the client certificate presented to the proxy in front of the web database has the right identity and then either allow it or not. So we only have to replicate that one rule around the cluster instead of 100 rules around the cluster. Yeah, this is kind of how we can scale that.

So let me circle back to your question, which was like why are we talking about service mesh when Consul does all these other things. The answer is that it's kind of the ideal platform if you like to build service mesh on to. So that's what we've wanted to do.

[00:36:16] JM: So I think it's worth talking a little bit about what we want out of a service to service communication proxy and the control plane that would sit across that service to service proxy communication layer. I alluded to this a little bit earlier, the fact that the service mesh term is used to describe all these different things that we want; load balancing, and metrics, and service discovery and all these different things. But what we need the most out of communication between our different services, and maybe just describe why services even need to communicate with each other in the first place.

[00:37:00] PB: I think that's an interesting question. I think it depends quite a bit on the team and the organization. I think one thing that kind of the service mesh and the market has focused on a lot so far is kind of very clever layer 7 features, like retries on certain error response codes and dynamic routing that allows you to move traffic from one service to another dynamically based on a percentage. I think that our organizations who this is really a helpful thing and it's really enabling them and they're kind of sophisticated enough and far enough down the path of microservices transformation to really use those things.

I think what we find kind of pragmatically speaking to a lot of especially larger organizations is that they're actually still quite a long way from this. In some cases, just having one tool knows where all of their services is is kind of a – That's quite a big idea and a new thing. So I think our goal with Consul, we're kind of working a lot on this kind of service mesh layer and that we think there is a lot of value there. But our kind of pragmatic approach is not that everyone should be running a service mesh. It's more that there's value at all of these different layers. For some companies, the value is just having one database of where all their stuff is running, even if they're not using it for discover. Even if they're still hard-coding IP addresses or kind of just using DNS from somewhere else. Just actually having all the stuff in one place is kind of early thing that Consul can offer them.

For some, just the discovery and being able to just make connections between the application instances based on DNS without having to go through a central load balancer for every single request. We've had some users who have like saved a ton of money because they used to have 2,000 firewall instances in their infrastructure, and now they've got rid of them, because each service is just using DNS to discover and speak to the other things it needs and they don't need anything more sophisticated than that for now. So Consul has a good story for that case.

So I think our approach is that there are lots of these different aspects as you say and different organizations at different points in their kind of journey in their maturity are going to need different ones. Our goal is to kind of have a toolset that lets you get kind of the best of all of the worlds.

[00:39:43] JM: Yeah, it's a pragmatic approach, which I admire and I like it. We talked a little bit about this in the interview I did with Armon and Mitchell. The focus of Consul right now as a "service mesh" is security, at least in terms of the newer features that you're working on. Explain what two services need to do to establish a secure connection with each other.

[00:40:11] PB: So that's a great question. So the security features that we've added, they've kind of loosely been grouped in with service mesh, but we'd like to talk about them in terms of segmentation. The problem they're solving in your network is segmentation. By that, I mean, restricting access to a resource to the like minimum set of other resources that might need to access it.

You can do that in traditional networks. You tend to do that with things like very tight V LANs around each service and type firewall rules that kind of only whitelist the IPs of this instance as you know. As we've just discussed, that's very difficult to scale. So the idea with Connect and the security aspect, the segmentation aspect, is to encode the identity of each service. So that means if you register a web service, the identity is just web. If you have a DB service, then identity is DB. It doesn't matter how many instances of that service you have. It's just the single identity. Encode that into TLS certificates, which is standard and kind of well-known way to secure your communication. Then have every connection be mutually authenticated.

So the web service presents a client certificate that says, "I'm the web service." The DB presents the service – Or the proxy in front of the DB presents their server certificate that says, "I'm the database." Then the destination is the one that checks with Consul. It doesn't just do the standard TLS handshake and make sure that it's a valid client. But then it makes a call to its local Consul agent and says, "Is the web service allowed to connect to me as a database or not?" So in Consul, you can have this central rule that says, "Web is allowed to talk to database." So that connection will be established. Then the packets will flow as normal over this TLS connection.

But if the intention graph, as we call it, like the list of rules about which services are allowed to communicate says, "No. Web is not allowed to speak to DB." Then the DB proxy just drops the connection on the floor and it doesn't let anything connect to the DB at all at the network level. So that's kind of the overview of MTLS between two services and how we want to kind of enforce them.

[00:42:35] JM: Let's talk about the problems of security in a large distributed system more broadly. There's been the rise of this term zero-trust networking. Can you define that term zero-trust networking as you see it and explain some of the solutions around providing a zero-trust model?

[00:42:57] PB: So the term zero-trust or some people prefer the term low-trust, but we'll use zero-trust for now. So the idea with zero-trust network is kind of a move away from the more traditional networking setup, which was that you have a secure perimeter, and that is like at the boundary of your data center where you're making one connection effectively where traffic is coming in from the outside. You have this secure perimeter, which is a firewall and a demilitarized zone. Then, basically, anything that gets into that gets in through your central load balancer or whatever it is, is essentially trusted and it's kind of a free-for-all and it's just easier that way, because you don't have to configure thousands of different firewalls with thousands of different IP addresses that change all the time.

Zero-trust networking is really about moving away from that model. The downsides of that model are hopefully fairly obvious, that if someone does get through your perimeter, then kind of the game is over and they can really do a lot of damage. The security principle of like – The principle of least privileged kind of is out of the window in that model once they're inside the network.

So the idea of a zero-trust networking is that you basically assume the attacker is already in your network at all times. Then ask yourself, "How do we limit what an attacker who's on this box has access to the absolute minimum we need to for this service to actually do it. It's not regular job." As well as that like it encompasses ideas of like if someone does compromise a particular node, how do we ensure that we can detect that and that we can rotate all the credentials that are on that – That the attacker has access to as quickly as possible so that we can really kind of limit the impact of that attack?

So with mutual TLS, the idea is that it's zero-trust and that even if the attacker is on the network, he's not – Or the attacker is not going to be able to connect to your database even, let alone guess your database password if you have that enabled, but they're not even going to be able to establish a network connection to it unless they happen to land on a box that already explicitly has access to it through, because there's an instance of the service that actually needs database access there.

[00:45:22] JM: We did a show about the Spiffy project, and the Spiffy project addresses one area of zero-trust by providing a standard for workload identity. Can you explain what workload identity is and how that relates to this problem of the zero-trust networking?

[00:45:40] PB: Yeah. So workload identity is very much what we've been talking about when we've been talking about service identities in this case. That's about traditional network security has been focused on IP addresses and kind of IP addresses and ranges are a kind of network identity, if you like. The problem with that is once your workloads become very ephemeral, maybe you're running in containers on a scheduler. Maybe you're kind of using software-defined networking and whatnot, and actually your service instances are moving around all over the place.

The idea that they're defined by one IP address is kind of eroded to a degree where it's actually not very helpful and a lot of the IP-based security tools just aren't very useful anymore in that world.

So the idea of service identity is that each service identifies itself not by where it's running in the networking, because that's not something we should concern ourselves with in the age of a scheduler. That's someone else's job to figure out where it's running. But by some kind of credential that form a trusted source can kind of claim to have a particular identity.

So, in fact, Consul Connect uses the Spiffy standard for the way we encode our TLS certificates to say, "This particular instance is a web service instance," for example. The way that that trust is boot strapped in Consul is that we have an ACL system, which you can set up in such a way so that the tooling that deploys your application is given a token that is specific to that service. So it can only deploy the service that has that name.

If you have a token that kind of grants you access to deploy that service, then you're able to get a certificate signed by Consul's internal CA, which grants you the ability to be that service, and it's only then that you'll be able to connect to other things that are allowing that service to connect to them. So like the idea of identity gets tied back to credentials that you can, as an operator, ensure or only deployed kind of alongside the actual code or process that really needs those credentials.

[SPONSOR MESSAGE]

[00:48:05] JM: DigitalOcean is a reliable, easy to use cloud provider. I've used DigitalOcean for years whenever I want to get an application off the ground quickly, and I've always loved the focus on user experience, the great documentation and the simple user interface. More and more people are finding out about DigitalOcean and realizing that DigitalOcean is perfect for their application workloads.

This year, DigitalOcean is making that even easier with new node types. A $15 flexible droplet that can mix and match different configurations of CPU and RAM to get the perfect amount of resources for your application. There are also CPU optimized droplets, perfect for highly active frontend servers or CICD workloads, and running on the cloud can get expensive, which is why DigitalOcean makes it easy to choose the right size instance. The prices on standard instances have gone down too. You can check out all their new deals by going to do.co/sedaily, and as a bonus to our listeners, you will get $100 in credit to use over 60 days. That's a lot of money to experiment with. You can make a hundred dollars go pretty far on DigitalOcean. You can use the credit for hosting, or infrastructure, and that includes load balancers, object storage. DigitalOcean Spaces is a great new product that provides object storage, of course, computation.

Get your free $100 credit at do.co/sedaily, and thanks to DigitalOcean for being a sponsor. The cofounder of DigitalOcean, Moisey Uretsky, was one of the first people I interviewed, and his interview was really inspirational for me. So I've always thought of DigitalOcean as a pretty inspirational company. So thank you, DigitalOcean.

[INTERVIEW CONTINUED]

[00:50:12] JM: Before I started researching for this show, I was under the impression that service mesh was something that Consul had been augmented to fulfill, because I was looking at this thing and I was like, "Okay, this is like five-years-old. They've been using it for service discover for a while and they just kind of augmented it to fulfill the service mesh category." But it seems like Consul has actually satisfied the requirements for "service mesh" since before service was really a term that people used. I guess the terminologies is really kind of – I guess it's semantic, because it does differ from company to company that has a service mesh to offer, like Kong.

I mean, I talked to Kong, and that's a company that starts as an API gateway, and now it's a service mesh. It's kind of like an avenue for product expansion rather than anything specific. But because service mesh is this overloaded term, it's this really heavy abstraction that can potentially cover all these different use cases. It makes sense for a lot of companies to build into it or build one in or have one separate it, and you can have this different thick service meshes and thinner service meshes in terms of how much work they're doing across your architecture.

So you've focused on the security side of service mesh, and maybe in the future you could get into load balancing or whatever that other people would want out of a service mesh, but the focus today at least is Consul Connect. It's the security side of the service mesh. Am I understanding the product strategy correctly?

[00:52:02] PB: Yeah, I think you are, and I agree to exactly what you said about it. It kind of depends whether you're using service mesh as this catch-all term for the space of like discovering and connecting and kind of – That's an infrastructure problem service mesh, which is how do all my services talk to each other? Then there's kind of this most specific definition, which is sidecar proxies and kind of the feature set that come with that.

So, yeah, and Consul as like broad problem has always been a problem. This has always been the problem that it solves, like how do I discover and connect to different services. I think, like you said, we're kind of trying to also offer this kind of sidecar proxy-based feature set that are people are interested in.

But you've picked up as well, as I mentioned earlier, that we actually built Connect first as a security product. The idea was less way want to be a service mesh. For that, we need proxies. It was more we want to secure everybody's workloads without them having to use their firewalls. So how can we do that? We can do that with MTLS, and that's why Connect – You can integrate your app directly with Connect to not have a proxy and still get all of the same security benefits. We can give you your certificate. We can automatically rotate it for you. If you make the right up calls to our API, we can have your application check, authenticate every inbound connection to see if it's actually allowed by our intention graph. You can do all of that in your application if you want to, which is not really the service mesh way.

But that's because we wanted to solve that segmentation problem with TLS first, and it's only kind of – As you kind of think through that line of thinking, you think, "Well, actually, in practice, 99% of people aren't going to retool their apps to use this thing," or at least not right away. So you're going to need some kind of proxy that sits next to their application that does the TLS part for them. That's why we launched Connect with only L4 connectivity.

Really, the main benefit you got from it was this security. The strategy, as you mentioned, the strategy is we want to kind of support wherever your organization is or whatever level in this kind of journey we've been talking about. We want to support that. So we are trying that. We are planning in the next few months to add some of our kind of L7 observability and routing and things like that that people are kind of aspirationally looking towards now. Yeah, recognizing that there's these kind of stopping off points all the way down the chain of valley.

[00:54:46] JM: So smart architectural and product design choice, and it makes me glad I did this show, because I think it has interesting contrast to all these other approaches, the Linkerd approach, and the Istio approach, and the App Mesh approach. It feels like a very HashiCorp type of approach, which I like. This issue that's interesting with the security side of things, which in a few previous episodes has been touched on. That's that security in one of these microservices architectures is often handled by this centralized policy management system.

If you have these services that don't have a service proxy or a service mesh layer, whenever they have to do policy lookups, they have to say, "Can I access this database?" They have to go hit this centralized big, bulky, policy management system. What you get out of the sidecar model or the service mesh model with the data plane s you've got this abstraction that sits local or very close to you and you can just hit up that module or that abstraction, that node, for local security policy and you can really cut down on latency in a relevant fashion.

Can you talk about that? That latency characteristics a little bit? The latency decrease that might be associated with adding this kind of sidecar proxy?

[00:56:19] PB: Yeah, certainly. I think there's an interesting subtlety in the way you describe that too, which is you get a latency decrease assuming that the policy is pushed down to the sidecar proxy. So one pattern that some service meshes have taken is to keep that policy kind of centralized. Then even though you have the sidecar, which is accepting traffic and is kind of load balancing your outgoing traffic, whenever a new request comes in, they actually have to go into an authorization call to a central service across the network to say, "Is this thing allowed or not?" That is obviously kind of the pattern you're saying that is difficult with these source of systems. I think most of the major service meshes are more sophisticated than that now, and to some degree or other, they either cache or they push things down into the proxy to try and reduce that latency.

I think it's an important problem though. Our kind of design goal with Connect was to really take advantage of the fact that Consul already has an agent running on your local machine. So the way that agent actually works in Connect is as soon as you register a Connect enabled workload on a machine, that agent, even before your proxy is running, that potentially has already generated a CSR. It's got it signed by the servers for the TLS certificate you're going to need and it's watching for any changes in the root certificate or for expiries. So it can automatically rotate it for you. It's also pre-fetched all of the like subset of the intention graphs, this thing that says, "Who's allowed to talk to who?" has pre-fetched that into local memory as well and it's pre-fetched all the service discovery results that you're going to need for all the things you've said you need to be able to speak to.

So when your proxy actually comes up, although that stuff isn't necessarily pushed into the proxy, it's all ready. So as soon as connections come into that proxy, it's going to make a call to the local agent, and the local agent is able to make their authorization call out of local memory and respond, typically, micro seconds of latency to kind of make those authorization decisions. Then we also perform all that authorization at layer 4. So identity is about who can connect to who, and we don't re-authenticate every single HTTP request that comes through.

In the future, there's certainly scope for having layer 7 rules about which identities are allowed to access which paths, for example. That can be enforced in the proxy, but the actual decision about which identity is allowed to connect to which other identity kind of at the network level is done there, which generally like amortizes the cost even further even though it's kind of only a local in-memory lookup.

[00:59:09] JM: Paul, this has been a really interesting conversation, and I feel like we've covered this service mesh category pretty deeply. I just want to close off by getting your perspective on the "cloud native" ecosystem and what you think will happen in the near future. There's a lot of change going on. There are cloud providers, container frameworks, service mesh stuff. A lot of business is being created, a lot of money being thrown around. What's going to change in the near future?

[00:59:39] PB: I'm not sure of my crystal ball is quite big enough to have all of those kind of questions. But I think one observation that I think is interesting in this space is that service mesh has a lot of kind of attention and mindshare now, but it's still really early. There are some people who are running in production. In general, a lot of the larger organizations are still trying to work out how they do kind of some of the more fundamental parts of their kind of transition to cloud and these sorts of things.

I think we're kind of far from done in terms of calling the winners in the whole space, but certainly in terms of service mesh. My feel is that businesses are going to get most value from solutions that kind of grow with them, kind of solve a problem for them now and go forward. Whereas, some feedback we've heard is that a lot of people kind of look at service mesh and play around with it as a proof of concept, and then kind of look at their actual thousand services that are in their on-prem data center on VMs, and it's just very hard to kind of plan this big bang transition. Everything is going to be in service mesh in a year.

So, yeah. I think growing with the organizations that are making this transition is going to be important. I think the other big thing is that organizations are increasingly making kind of multiple cloud selections, and I think this is inevitable both from a disaster recovery, like not putting all your eggs in one basket. But I think more likely than that, just large organizations just have teams with different needs. They want to make the most of different features that different cloud offerings provide, or even the strictest organizations probably will end up bringing in infrastructure for mergers and acquisitions. That means they end up with a footprint in lots of different cloud providers.

So having solutions that are kind of independent from those providers and are not tightly coupled to just one provider are going to be important. I think there's one other reason we're seeing such as interesting Kubernetes. But I think it's less obvious in some of the other categories kind of how you kind of play that game. I think that's one of the things HashiCorp is kind of very focused on across our suite of products is how do we provide a really great workflow and a great solution in the problem space, but provide in a way that's going to work really well for you whether you're still on-prem, whether you're in multiple clouds and kind of into the future.

[01:02:10] JM: Well, Paul, thanks for coming on Software Engineering Daily. It's been great talking.

[01:02:13] PB: Thank you.

[END OF INTERVIEW]

[01:02:18] JM: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can build data-driven websites and professional web apps very quickly. You can store and manage unlimited data, you can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your sites functionality using Wix Code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix codes, built-in database and IDE, you've got one click deployment that instantly updates all the content on your site and everything is SEO friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There's plenty that you can do without writing a lot of code, although of course if you are a developer, then you can do much more. You can explore all the resources on the Wix Code's site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix Code experts.

Check it out for yourself at wicks.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you can do with Wix Code today.

[END]