EPISODE 611

[INTRODUCTION]

[0:00:01.3] JM: Rust is a system's programming language with a distinct set of features for safety and concurrency. In previous shows about Rust, we explored how Rust can prevent crashes and eliminate data erases. Through its approach to type safety and memory management. Rust's focus on efficiency and safety, makes it a promising language for networking code. Tokio is a set of networking libraries built on Rust. Tokio enables developers to write asynchronous IO operations by way of its multithreaded scheduler.

Tokio's goal is to make production ready clients and servers easy to create, by focusing on small, reusable components. Carl Lurch is an engineer at Buoyant, a company that makes the popular linker D and Conduit service mesh systems. Kubernetes developers deploy service mesh to their distributed applications as side car proxies.

These proxies need to be low latency and highly reliable. In that light, it makes sense that Conduit which is the more recent service mesh from Buoyant is built using Rust. Carl joins the show to describe why Rust is useful for building network services.

[SPONSOR MESSAGE]

[0:01:24.3] JM: At Software Engineering Daily, we have user data coming in from so many sources. Mobile apps, podcast players, our website and it's all to provide you, our listener, with the best possible experience. To do that, we need to answer key questions like, "What content our listeners enjoy, what causes listeners to log out or unsubscribe, or to share a podcast episode with their friends if they liked it?"

To answer these questions, we want to be able to use a variety of analytics tools, such as mixed panel, Google Analytics and Optimizely. If you have ever built a software product that has gone for any length of time, eventually you have to start answering questions around analytics and you start to realize, there are a lot of analytics tools.

Segment allows us to gather customer data from anywhere and send that data to any analytics tool. It's the ultimate in analytics middle ware. Segment is the customer data infrastructure that has saved us from writing duplicate code across all of the different platforms that we want to analyze. Software Engineering Daily listeners can tray Segment free for 90 days by entering SE Daily into the "How did you hear about us" box at sign up.

If you don't have much customer data to analyze, Segment also has a free developer edition. But if you're looking to fully track and utilize all the customer data across your properties to make important customer first decisions, definitely take advantage of this 90-day free trial, exclusively for Software Engineering Daily listeners.

If you're using cloud apps such as MailChimp, Marketo, Intercom, App Nexus, Zen Desk. You can integrate with all of these different tools and centralize your customer data in one place with Segment.

To get that free 90-day trial, signup for Segment at segment.com and enter SE Daily in the how did you hear about us box during signup. Thanks again to Segment for sponsoring Software Engineering Daily and for producing a product that we needed.

[INTERVIEW]

[0:03:55.0] JM: Carl Lurch, you are an engineer at Buoyant, welcome to Software Engineering Daily.

[0:03:58.9] CL: Thank you, how's it going?

[0:04:01.5] JM: It's going quite well. Today we're going to talk about Rust and specifically, networking in Rust. We have done a few shows on Rust, we did a show on the language itself with Steve Clabnick. He was a fantastic guest and then Alex Crichton came on the show to talk about Rust concurrency. Those episodes might be a good preface for people who are completely unfamiliar with Rust.

But even if people aren't unfamiliar, we should give them a chance to catch up. Let's start with some facts and concepts within Rust. Rust is widely used as a systems programming language. What are the advantages of Rust as a system's programming language?

[0:04:41.5] CL: First, systems programming language can mean different things to different people. The way I kind of like to describe it as you can write an operating system or device drivers or like maybe even embed Rust in other programming languages. For that to be the case, you cannot have a run time. I will get to that probably in here but like Go Java, all those use run times.

The traditional systems languages have been C and C++. With those languages, if you're not familiar with them, you have to manage your memory yourself, you're dealing with pointers and they're known to be, let's just say, unsafe. Since the programmer is responsible for ensuring that their pointers are always pointing to live memory and to the correct location because the programmer is responsible for allocating memory and freeing it at the right spots.

It opens up the programmer to writing bugs that can have like very severe consequences in production. I guess a lot of the vulnerabilities that have gotten a lot of publicity have been done to programmer's writing bug, I know it's rare for us programmers to write software bugs, especially when managing memory but it can happen.

Rust on the other hand is kind of targets the same demographic in C++, it offers, like, when you write a Rust program and you compile it, so it is compiled, right? You compile it, the end executable, the end results is something as roughly equivalent to what you would get with C or C++.

It can run in a lot of the same locations. The big difference is that when it compiles, you're guaranteed to not have any memory vulnerabilities because the Rust language itself manages the memory for you.

It's able to do this entirely at compile time. That's the big thing, that's like really the big new thing that Rust brings to the table. It's a great language for many reasons besides that but it has tons of really great language features that make it ergonomic and just fun to use but I would say, like

a lot of those features aren't like, unique to Rust. I bet a lot of those things like House Call people will be like, "Hey, we do it first" and then someone else before them will say, "Actually, we did it first."

But the way that Rust gets you like managed memory, I guess the Rust language manages the memory for you but entirely at compile time, that is something that is very new, and I mean, as far as I know, Rust basically did it first outside of like the academic realm.

Other Rust developers probably know more about it but like this definitely came out like academic research and if only there were some toy, like not toy, like academic research languages that exemplified the things that Rust is building on but Rust itself is kind of the first language that has been generally used in industry that offers these capabilities.

[0:07:52.2] JM: Those capabilities you're referring to, are those what give Rust the safety, because it's often described as a safe programming language?

[0:08:01.3] CL: Right. I guess the way to think about like, if you were doing it on C or C++, right? You would have like, I'm going to allocate some memory and then you get a pointer to it. Then you pass that pointer off to some other location in the code. Whatever gets that pointer, it needs to assume, because it didn't allocate the memory, something else did and something else kind of, is responsible for freeing it but whenever you pass pointers around, those colocations have to assume that at that point, the memory hasn't been freed and it's still valid.

Because in those – when you just have a pointer, you have no real way of knowing if the memory is still valid or not, so you just kind of have to assume that whatever gave you, that pointer is doing the right thing. Now, if you – especially in the face concurrency which is probably why like Alex didn't – was able to like give a good podcast on it but like, especially in the face of concurrency, ensuring that your pointers and the memory stays valid gets really complicated.

It's very easy to say, "While this other part of the code base is using the pointer, assuming that's still valid, I accidentally freed it too early" and then that memory location gets reused or the

memory location gets unmapped, like any number of bad things can happen at that point, resulting in site faults or even worse, memory vulnerabilities.

The main kind of thing that Rust does for you is like, when you allocate memory in Rust, the compiler's able to keep track of exactly where in the code that like that memory is owned. In Rust, every piece of memory has one owner, right? It's able to statically see where as ownership is passed along different variable or locations, so if the owner says, "I don't need it, I'm going to pass it off to this other variable," then it compiles able to see, okay, "This variable doesn't own the memory anymore, now this new one does," right?

Because it's able to statically track that at one point in your code base, that variable, that's owns the memory, it goes out of scope, without handing off ownership, the Rust compiler can see like, "This is where like the memory can no longer be accessed anymore" because it can statically track every point in your code base that accesses the memory.

Once it sees this memory can't be accessed anymore, it frees. It fees it at that point because it knows, nowhere else in your code base can you access that memory. Now, of course, having just one variable that can access a piece of memory is not super useful, Rust also has the concept of like borrowing.

An owner can maintain ownership of a piece of memory but then it can like say, "Call f unction say actually, I'm going to let you borrow this memory for as long, for the light time in this function call," kind of thing.

The memory stays at the original location but another part of the code can borrow it. Now, the thing there is what Rust can do is say, "Okay, I can guarantee a compile time that the owner of that memory stays around longer than the piece of code that borrows it." That kind of ensuring that the memory stays live as long as it needs to, is basically what Rust does.

That's being able to do that kind of carries over like, I mean, at a very basic point, it just manages the memory for you, preventing all these like security vulnerabilities that come along when you make mistakes in that area.

Then you can take those capabilities and carry them over to like other kind of concepts like concurrency and use those kinds of features around ownership to model API's for concurrency, for network programming, for like all these other things that gets the programmer additional safety by putting more heavy lifting on the compiler, if that makes sense.

[0:11:56.4] JM: It does. Earlier, you said something about a term called a language run time. What does it mean to have a language run time?

[0:12:07.3] CL: Language run time is any kind of library code that the compiler has to inject into, or not a silly inject into the index executable but somehow, when a compiler takes your code and compiles it down to a target, the run time is anything that that end target requires to run.

For example, at the very basic, even C or C++ has like a run time but in their case, it's going to be extremely small. For example, when you call mal lock or Lib C is kind of like those C run time or C++ has its own runtime that it uses.

I don't even know exactly what I believe it like for unwinding and like exceptions and those kinds of things. Go on the other hand, like is also a compiled language, it does not need a virtual machine but it provides like, I mean, if you're not familiar with Go, it gives you a lot of capabilities like garbage collection, green threading, go routines, all those kinds of things.

It gets you all those capabilities by having a fairly large run time and the run time again is just any library code that is required for the language to run once it's compiled.

[0:13:25.4] JM: Okay, we had this show with Alex that we've mentioned a couple of times now with Rust concurrency and he describes some of the concurrency features of Rust in detail. As you said, before, it's useful to have these lower level safety properties because when you have safety at a low level.

If you build things on top of that then the safety commutes to the concurrency primitives that you build on top of those safety primitives. What are the concurrency primitives that Rust offers?

[0:13:58.9] CL: Rust itself is a language and as the standard librarian, what you really get when you just use Rust is pretty much just a language over anything that the operating system gives

you. Okay, basically, that means that you can get threads. Threads is an operating system construct and that's roughly like the foundational of its concurrency story. I mean, the standard library gives you other tooling like atomic reference, counted, cells and channels for passing data between threads but the main concurrency pattern is just using operating system threads.

That's at like as supposed to for example, Go, gives you green threading so that's kind of – it has its own concept of what a thread is for the language and a compile time. It then takes all those language threads and implements its own scheduler and multiplexes of them on operating system threads.

When you're just using Rust, straight out of the box, you get operating system threads and that's basically it. However, because Rust is a systems language and not being able to build in Rust itself.

Anything that you can imagine but anything at any other language implements, you can basically implement directly in Rust because Rust is a language of the level that you could use it to actually like implement Go, you could use Rust to implement other languages, right?

Because it's at the same level of C or C++. Once you get at that level, right, you can maybe not necessarily bundle a lot of things that are offered kind of as part of the language and higher-level languages. You can just kind of offer this like core primitive and then let the higher-level capabilities like, well, network, like asynchronous none blocking networking and anyway, we're going to get there but you can let – you can leave all of those things to libraries.

Which lets the end user pick and choose what works best for their case, like for example, if you're implementing something to run on a large server – you're going to have a completely different set of requirements than if you're doing embedded programming on a tiny like mobile chip, right?

Rust lets you target both of those use cases and lets libraries build up their ecosystems outside of the core language so that all these different cases can be best served like at the library level.

[0:16:32.8] JM: Tokio is something that you have worked on a lot. Tokio is for networking in Rust. Explain what Tokio does?

[0:16:45.6] CL: Tokio is built on operating system primitives, it provides asynchronous networking and IO. Like roughly, it gives you the ability to program with TCP, UDP, mixed domain sockets, it also gets you file IO and a bunch of other kind of stuff that you would expect at the IO layer and it uses operating system primitives like on Linux, E Pole on like OS 10, KQ, BSD's KQ, Windows is IOCP, etcetera, right? It takes all of those different none portable API's, builds a shim layer on top of it, so actually like at the very low level below Tokio is actually MIO which is at the absolute lowest you can go, it builds a compatibility layer on top of those different operating system primitives.

Tokio is a higher level on top of that which gives you a more, a richer kind of full featured asynchronous environment. If you're implementing like high performance servers or clients in Rust, you would be able to use Tokio to kind of get you everything you need out of the box.

For example, no JS or Go. Those actually provide a lot of those primitives as part of the language. To get an experience like Node or Go, you would probably use Rust plus Tokio.

[0:18:11.5] JM: I could use Tokio to build a web server but Tokio itself is not a web server?

[0:18:17.3] CL: It is not a web server, for example, what it gives you is asynchronous TCP, asynchronous UDP API's, it actually gives you a scheduler because it has a concept of a task which is similar to an air link process or a go routine but it's asynchronous. There's no like green threading, it's kind of like a concept of like a unit of execution but for asynchronous and there's schedulers to take all of these different tasks and multiplex them over like a thread pool.

You can implement a server using like these primitives that Tokio gives you. Break it up into like lots of little tiny tasks, so for example, you would write one task web handle one specific socket.

You would then give all those to the Tokio scheduler and it would handle multiplexing all of that across like a thread. Doing the traditional MN and scheduling pattern. You have N tasks, they're

probably a very large number of them, probably reach the millions and then you'd have like eight threads and scheduler handles scheduling all your task across all those threads.

[SPONSOR MESSAGE]

[0:19:40.5] JM: Every team has its own software and every team has specific questions about that internal software. Stack Overflow for teams is a private secure home for your team's questions and answers.

No more digging through stale Wiki's and lost emails. Give your team back the time it needs to build better products. Your engineering team already knows and loves Stack Overflow. They don't need another tool that they won't use. Get everything that 50 million people already love about Stack Overflow in a private secure environment with Stack Overflow for teams.

Try it today with your first 14 days free, go to s.tk/daily. Stack Overflow for teams gives your team the answers that you need to be productive. With the same interface that Stack Overflow users are familiar with. Go to s.tk/daily to try it today with your first 14 days free.

Thank you Stack Overflow for teams.

[INTERVIEW CONTINUED]

[0:20:55.0] JM: In the situation where you have a million tasks to schedule and you are scheduling them across eight threads. Could you give a more concrete situation? What is going on in my program that would lead to a situation where there are a million tasks and eight threads to be scheduled on, to help people map the idea of tasks to higher level program execution.

[0:21:18.1] CL: Let's start very simple, right? Let's say you're implementing a Hello World server. You're implementing a server that just accepts sockets and then writes Hello World to them and closes them, right?

One way you would model that like with – let's start with threads, right? If you're building like a very like traditional blocking, like synchronous system and you get threads for concurrency, you would open your TCP accepter, listen for sockets on it and now live on a thread. Whenever you accept a socket, you would spot a new thread and say, "That thread handles that socket," so you would end up getting one system thread per socket, right?

Now, without trying to start a flame war about threads versus non-blocking, let's just say, best practices tend to shift towards using less threads than one per socket, right? Because native threads tend to have upper limits in the number that you really can have simultaneously. If you want to get into the millions, you're going to probably not want one thread per socket. The way you do that is, if you were using E-poll directly, as you would have one thread and you would use the E-poll primitive to get, to receive events.

E-poll just tells you, okay, this socket is ready to do something with. You would ask the operating system, "Here am I, million sockets, let me know when one socket is ready to read data from" and E-poll would just sit there and then some sockets are ready, it would say, "Hey, these are the sockets that that are ready" and on that thread, you would then loop through all those sockets and then write some data to it.

That's a fine way to do things, it's just is a little burdensome to write your server at that low level. Tokio comes along says, all right, we're giving you the concept for a task which roughly speaking of, you can think of as you can just take a unit of logic from your application in the case of the Hello World, that would just be one unit of logic is like the task of accepting new sockets from the listener, right?

Then you just have to spawn one task that says, just loop over this, XTCP acceptor and when you get a socket, take that socket and spawn a new task. Just like you'd spawn a new thread, in the blocking model, here you just spawn a new task and then that's new task that handles that one socket. It just reads data from it and writes data back to the socket.

Now, in this kind of very simple model, you get one task that handles accepting sockets and then one task for every single socket that you accept. By breaking it down into these kind of

individual tasks, it's like more manageable, you can reason better about it and on top of that, the scheduler can do smarter things with it.

When I describes with the raw E-poll, I just said on one thread, you just loop and handle all of them. But now if you want to start adding multiple threads because you want to add some concurrency to the story because if you only use one thread, you can only use one core.

One physical CPU, if you want to start using more than one physical CPU because modern day servers are stealing out in terms of number of CPU's, you need to start thinking about "Okay, how do I take my application and best utilize all the physical resources available to me."

By breaking your application down to small units that are fairly isolated, in this case, one task for one socket and you give them all to Tokio. Tokio looked at all of those tasks and says, "Okay, how can I run this best across the physical CPU's that are available to me?"

[0:24:59.7] JM: Tokio is an entire ecosystem. There are many different things you could build with Tokio, what else needs to be within that ecosystem to enable people to build tools for networking with Rust?

[0:25:16.1] CL: Two thoughts come to mind when you say that, yes, first of all, Rust and the networking and Tokio and the networking ecosystem is pretty young compared to other languages. We're still building out, well, there is an ecosystem, you are correct, we're still building it out, it's not as established as like one with Java or Go would be. But also like, in the Rust community, there tends to be a philosophy of breaking down libraries.

Kind of like the micro library approach which I don't really love the term but it kind of fits, where you break down the abstractions into like smallest unit, that makes sense together. That makes sense staying alone, right? When I talk about Tokio, it's kind of this collection of libraries that fit together. Some are maintained by me and some maintained by other groups.

For example, roughly speaking, all the abstractions around asynchronous programming are actually maintained by the Rust team roughly in a crate, in a library called Futures. A very creative name, it just provides futures and the asynchronous model.

Tokio takes that and builds on the networking layer. Then, when I first started working on Tokio, we were like, “Okay, let’s have Tokio be everything.” Anyway. Since then, we’ve been focusing to make Tokio exactly what I describe earlier but besides that, that just gives you as in other languages, that just kind of gives you the lowest level primitives and you’re right, you want to build like web frameworks, you want to build like higher level primitives for writing your clients and servers so that you don’t have to reinvent the wheel every single time, right?

Still, thinking on at the raw socket level is still pretty low level. Another project that I’ve been working on that’s still very early, it’s not even released to crates.io yet, it’s called Tower. It’s something that we’ve been working on at Buoyant for Conduits, which I’m thinking we are going to get into it in a bit. But it’s very heavily inspired from Finagle and it kind of gets you a higher level of framework to implement your service and clients in, it’s really oriented around the request-response model. Like I said, it’s very similar Finagle, or also like Rack and Ruby although that’s only HTP. Towers like HTP, like any other protocol.

I think like Express JS is also kind of like abstraction around request response for node but we’re kind of building that out separately. The end goal hopefully at some point is that you’ll be able to write a web service or a GRPC service as easily as writing – in a way that you can just focus on your application logic if that makes sense. So there’s Tower which is the request response, there is Tokio which is just a runtime and then there’s people building other things on top of Tokio too. I think there’s Actics which is an accurate model. There is people building other RPC libraries with them. I think there was a – what is it called? I don’t remember the name.

There is Tarp, there we go, but yeah, there are people building higher level things on top of Tokio.

[0:28:42.2] JM: Taking Tower as an example, how have you used Tokio to build Tower?

[0:28:48.4] CL: Tokio since it just provides the sockets essentially, it provides the sockets and the run time like you break things in terms of tasks and you use sockets and it runs it. So with Tower what we are trying to figure out is, “Okay, given that how can we model a task to instead

just be responding to our request?" So can I a task just be, "I get a request, how do I generate the response to it?" And then from that we're building it. So we built an HTP2.0.

Well there's also Hyper which is 1.0 and 2.0 but there is HTP, there's GRPC. We have Tower GRPC which gives you a GRPC layer on top of Tower and again you could model since everything is just request response. GRPC is extremely request response oriented. So you just implement your end points as like, "Here is my GRPC request and let me return the response," and then what Tower does is takes that and breaks it down, like that server that you implement over client.

And breaks it down into tasks and submits it to the Tokio runtime, just run across all your CPU's kind of thing. Then of course, just like Ruby Rack or Express or Finagle – if once you model your applications in terms of request response, you can then build middleware which a middle piece of middleware is just something that takes a request and handles the response by calling some other middleware. So it just builds a stack where each layer gets request.

Does something with the request and passes it onto the next level and that level does the same thing all the way up, gets the response all the way down and by breaking it down you can say, "Okay my application. I am going to break it down to a stack middleware. One middle ware is only going to have handle retry." So that middle ware gets requests. It tries to pass it onto the next layer. If that layer fails and then the retries middleware's job is to say:

"Okay that layer had failed. Let me wait a second and then try again." Then now if you build an application on Tower, you could just say, "Okay all I am doing is implementing my application logic. I get a request and then I return a response but now, I can inject this retry middleware and just out of the box now that piece is going to handle retries for me." I just implement request of my – I get a request to response. If that processing fails, the retry middleware just tries again.

And other middleware might be load balancing. So that load balancer takes him gets a request and then distributes that request over any number of inner number of services. So the gist of that is basically you've got a very clean abstraction where you take a response, get a request which are response. By doing that you can break up your app into a whole bunch of loosely

couple of components and some of these components are very reusable and this was Finagles prompt.

Finagle's premise is like, "You just want to implement your application code without having to deal with all the gory details of making it production ready. But you can implement all the gory details in making production ready decouple from the application, even decouple from the protocol," but that is the rough premise. It's still very early on for Rust and like I said, we are building it for Conduits which is like a service mesh load balancer proxy. We're using this real-world production use case to build out this library that is useful to the general Rust community.

[SPONSOR MESSAGE]

[0:32:37.9] JM: Users have come to expect real-time. They crave alerts that their payment is received. They crave little cars zooming around on the map. They crave locking their doors at home when they're not at home. There's no need to reinvent the wheel when it comes to making your app real-time. PubNub makes it simple, enabling you to build immersive and interactive experiences on the web, on mobile phones, embedded into hardware and any other device connected to the internet.

With powerful APIs and a robust global infrastructure, you can stream geo-location data, you can send chat messages, you can turn on sprinklers, or you can rock your baby's crib when they start crying. PubNub literally powers IoT cribs.

70 SDKs for web, mobile, IoT, and more means that you can start streaming data in real-time without a ton of compatibility headaches, and no need to build your own SDKs from scratch. Lastly, PubNub includes a ton of other real-time features beyond real-time messaging, like presence for online or offline detection, and Access manager to thwart trolls and hackers.

Go to pubnub.com/sedaily to get started. They offer a generous sandbox tier that's free forever until your app takes off, that is. Pubnub.com/sedaily. That's pubnub.com/sedaily. Thank you, PubNub for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:34:21.1] JM: And so the real world production use case you would describe as Conduit and the more general library is Tower.

[0:34:29.1] CL: Right because without trivializing Conduit, there is actually a lot of code that is not in Tower. There is a lot of TLS security authentication, configuration like details on how to inject in your Kubernetes cluster. There is a ton of code but it is also a lot of it is just taking all of these different components like load balancing, retry, service discovery, buffering, how do you add buffering and that kind of thing. Taking that and gluing all of those different Tower pieces and gluing it together into a production stack.

[0:35:00.0] JM: So if I have an application that I want to be expressed in terms of services, Tower is useful for giving me the API, the interface for expressing in terms of a service.

[0:35:16.8] CL: Right, so unfortunately I have come to realize that service is a very overloaded term. So what Tower calls a service is actually pretty different from what Conduit calls a service. Unfortunately, I ended up at a company that merge, uses both terms liberally which makes it hard to talk about.

[0:35:36.5] JM: So maybe we could talk about service as vocabulary within the world of tower since it is a little bit lower level and then you could talk about how service is expressed in the context of Conduit, for the purposes of disambiguation.

[0:35:50.2] CL: So what I call a service in Tower is just a Rust traits called service and for those not familiar, a trait is kind of like an interface basically. It is a set of functions that you implement and then you can kind of operate on it generically. A trait is how you write generic code in Rust basically. Tower provides a service trait and I think it is pretty much just defining a function from a request to a response. So in Tower, calling something a service is just a fancy way of saying:

“It is a function that takes a request and a response” and when you think about it like if you are familiar with Rails for example, when you write an action like a controller action that single action is just a function that takes a request and returns a response. The same thing I’m sure applies with Go and there’s probably this abstraction where at some point, you write a function

that takes a request and you return a response. So all these fancy talk of services, for Tower it is just that. It is just a function.

[0:36:59.1] JM: Okay and now that you have given a little bit of color for how Tower works, can you explain how it is useful for Conduit?

[0:37:06.8] CL: So in the Kubernetes world, a service is going to be essentially a network process that handles requests and responses for one kind of small unit of logic because that is a micro service. So for example, if you are implementing some large application that's virtually an e-commerce site that sells widgets, you are going to have your front end web server that you get requests for widgets and then it's like, "Okay, you are going to try to buy this widget. Let me first authenticate you."

And to do that, instead of having all that logic internally in this one application. I am going to have another network process somewhere else in my infrastructure, that job is only authenticating users. So what the front end does is it makes a request to authentication service. So that service it just gets requests for authenticating user and returns a response that either authenticates or rejects, right? So, at a very high level, it is a similar concept and now you are getting a service.

Gets requests and returns responses but it's like apps, the network infrastructure process layer, right? So then again, your widget e-commerce will then make a payment request, a billing request to your billing service. Again, that goes over the network that thing that that network process was just focused on that one thing. It tries to process a credit card and then returns a successful response. So the main difference here is you're breaking up a large application into units of code.

That can be managed at an infrastructure level individually, so that you can as an organization whatever company is writing this widget e-commerce site that they can have different teams. So engineers make small like – you can have smaller teams of engineers focusing on one service and they could own it themselves. They can manage deployment, they can manage the development life cycle. They can manage deployment and all of that stuff as a team versus

have other services within the organization, they can be customers of them, if that makes sense.

[0:39:31.1] JM: Yeah and this functionality was in Linkerd to some extent which is the service proxy that Buoyant launched with. Conduit is the newer service proxy or service mesh depending on what you want to call it and it's in Rust as we've been discussing. When you think about the end to end networking stack for Linkerd which is in Java and you compare the overhead of Linkerd to that of Conduit. Conduit is less and why is that? Why is Rust a better choice for a service mesh than Java might be?

[0:40:16.0] CL: When you are taking a large monolithic application, breaking it up into network services like we just described, you solved one problem which is organizationally at the company. Now small teams can manage small things but you introduce a whole other set of problems which is now, when you want to do billing you don't do a function call internally, you have to make a networking call and as we all probably know, the network is not as reliable as if we are making a function call.

So to in order to maintain really robustness and production grade when you make that networking call, what you might do is insert a service mesh and that the job of the service mesh is to facilitate the communication between all of your different services to make it as easy for the developer to treat it like almost like a function call within the process, right? So the goal is really to make that networking call as robust as it possibly can.

So Linkerd was the first production grade service mesh that was released a number of years ago and it really started taking off before – as Kubernetes had also started wrapping up. So the way it was designed is you are going to dedicate some medal. So basically, you are going to take some heavy servers and you are going to dedicate those servers to just run Linkerd and then all your other application nodes or micro services they'll talk on their –

They'll live on their service and then when they want to talk to another service like they will talk to – they will make a network call to Linkerd on it service and then Linkerd will handle, do all its magic and proxy it to the call to the end goal, right? So the key here is that when it was

architected it was designed to one not be tied to Kubernetes. It was designed to handle all the flexibilities needed by existing organizations and it was designed to run in its own hardware.

And when it was built, it was built using Scala and Finagle. So the JBM is like a really great tool for writing production grade networking services but it tends to come with a large hardware requirement. So when you deploy a job app, you have to give it a lot of memory and a lot of CPU, like a lot of memory, and that's fine when you are dedicating a server to run that one Java process but once Kubernetes came along, it launched this whole other model or way to do things, which is the sidecar model.

Kubernetes handles running all of your services across all of the infrastructure and then what you do is you inject your service mesh process one time on every single pod. So a pod is just a group of services running on the same server. So the Conduit service mesh ends up running essentially on every single server you have in your infrastructure. Now you can use Linkerd and the JVM to do that but that's going to require a significant number of resources.

I think people actually working on Linkerd made a heroic effort to try to reduce the footprint of Linkerd, like the memory footprint, to get the very minimum smallest possible amount that you needed to run Linkerd. I think it got down to 100 megs of memory which to run a JVM process that is pretty amazing but still that doesn't really work right in the sidecar model. So Buoyant was looking at this problem.
They're like, "Okay we need to kind of – the sidecar model is great. Let's adopt it. What are we going to do? We're going to have to probably pick a different language like Scala to write our service mesh in." So at that point you're going to get down to, "Okay well what are we going to use?" We can use I think like Go probably has a small enough footprint. You've got C or C++, when end to disk service mesh is really a proxy like a network proxy.

You are going to run every single bit of data that goes over the network here, we'll run it through this proxy, so you wanted to be as fast and light weight as possible. So historically up until Rust started really picking up steam, you had C and C++ and then in the medium-term pass, you have Go. Go was pretty light weight in terms of memory, but it came with a bunch of runtime cost over C and C++. If you run an application in Go it is not going to be as light weight as a well-tuned C or C++ application.

But then historically you either had to decide “Okay, am I going to use C and C++ and get as lightweight as possible?” But open up the application to this whole category of memory security vulnerabilities that you get when you write memory management bugs and it is not a question of if you are going to write those. It is a question of you will write them, how fast will you discover those bugs and how many security vulnerabilities are going to be exposed to the world, right?

You are always going to have some level or some number of them or are you going to pay a little like some cost, runtime cost and use a language that is safe in that regards like Go. Go, by using a garbage collector and with the runtime that it provides, it is able to ensure memory safety in the same way that Rust does, but it requires some runtime costs to do so. Now the good news was when Buoyant was thinking about well what language should we pick.

Rust was starting to pick up steam especially in the networking A-synchronous networking layer of things. So now, there is this new world in which you can get the best of both worlds. You can get the low levelness of C or C++ plus the memory safety of Go. In fact, you actually get a little bit more safety because you can leverage the type system that design API’s in there that are like harder to misuse. So when Buoyant approached that –

I mean I am a little biased because I have been using Rust for I don’t even know, four years now? I don’t actually know, a long time in the world of Rust users. I think they made the only choice that you really can these days, like when you need the low-level performance, like when you really need every kind of little bit of juice matters because you are writing a proxy, Rust is really the only choice you have now. The only good choice, again I am biased.

[0:47:01.1] JM: Well it sounds like there are obviously a lot of strengths. There are also some interesting challenges because as we’ve explored from this conversation, you’re simultaneously working on Tower and Conduit, partially because Tower doesn’t exist in the Rust ecosystem. The Rust ecosystem is nascent. So you simultaneously have to build up the Rust ecosystem while you are building up an application level product.

[0:47:29.6] CL: Right, I mean it definitely was a gamble but when you think about it, for Buoyant for running a service mesh – the service mesh is the fundamental product that Buoyant is

building and Buoyant wants to go to the organization and say, “Hey run every single bit of data that you have through our product,” right? I mean again, the organization will manage it and have full control but you are adding this new wheel.

Like I said, if you pick C or C++, you’re going to end up with security vulnerabilities. That’s just fact. I am not aware of anyone who has built a significant networking product that has been exposed to the world, that does not have in C or C++ and has not introduced security vulnerabilities. In Buoyant’s choice they was like, “Okay let’s just say it’s critical for the users of our data to feel safe” I guess. “We want to minimize their risk and maybe even minimize it even further than if they used Conduit.”

Like reduce the risk even without Conduit in terms of potentially having security vulnerabilities and I mean let’s just say that data security seems like as the years go on seems like a more and more critical thing for organizations to really care about. So in that way like Go, C and C++ wasn’t even a realistic option for them. Well us now like they made this choice before I joined which is why I say them, right? Because I work there now but anyway it was basically between Rust and Go.

The question was Go was more established in terms of the library ecosystem. Rust was more nascent but offered a lot more in terms of performance and light weight and low memory capabilities. I think while Buoyant has to invest a lot in building that ecosystem up, one thing is we are building a proxy and that is pretty low level. So as the applications has been built from those features we are just thinking, “Okay how does it fit to rewrite directly in Conduit or should it exist in Tower?”

So we are building up the product and all we have to decide is well, do we want to keep it locked into Conduit or do you want to make it available to the greater world? So since these features that we’re building for Tower is actually pretty core to the value prop of the product, I think – I mean I am not a product person. These are all product decisions but my take on it is that it makes sense for Buoyant to actually have a significant amount of expertise and a stake in building these things.

Exactly in a way that it makes sense for a proxy. So one thing that historically when you are building like Finagle or these kind of abstractions like Tower, is they tend to exist in languages that already have a runtime cost and when building these abstractions, historically have focused more on, “Okay, we can add a little bit of runtime cost to the request response abstraction because it is just going to be running in an application layer and the application layer is going to have some much overhead that the little bit we add in the request response abstraction library doesn’t really matter.”

That is great there if you’re building it for a proxy where literally the entire application like the entire hot path for the data is through this request response abstraction, like every single bit of performance there is going to matter. So when we are building tower because Buoyant has this stake in building out that ecosystem, we can take the priorities that are important for Conduit as a product. So really, the short version is, I am just an engineer there this is my opinion and I am not a product person but I think the decision made sense.

[0:51:30.4] JM: Well Carl, I want to thank you for coming on the show. You have given a pretty thorough front to back understanding of why Rust is useful, what network abstractions have been built on top of Rust and an example of an application level product that can be built on top of those networking abstractions. While simultaneously helping to build up those networking abstractions to being more useful. So thanks again for coming on.

[0:51:57.1] CL: Yeah, thanks for having me.

[END OF INTERVIEW]

learnings of the ThoughtWorks engineering team who have talked about building the product in previous episodes of Software Engineering Daily.

And it is great to see the continued progress on GoCD with the new Kubernetes integrations. You can check it out for yourself at gocd.org/sedaily and thank you so much to ThoughtWorks for being a longtime sponsor of Software Engineering Daily. We are proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]