EPISODE 379

[INTRODUCTION]

[0:00:00.8] JM: Containers are widely used in projects that have adopted Docker, Kubernetes or Mesos. Containers allow for better resource isolation and scalability. With all of the adoption of containers, companies like Red Hat, Google and CoreOS are working on improved standards within the community. Standards are important to this community because its pace of growth and the number of concurrent projects could lead to chaos otherwise. If you heard our recent episode about the Linux kernel's open source governance, you know that having some rules in place will help encourage the right kind of creativity to thrive.

In the world of containers, networking is not well addressed because it's highly environment specific, the Container Networking I nterface is an effort to add specifications around how networks of containers can form. Dan Williams is an engineer at Red Hat and in today's episode, he explores the ideas behind the container networking interface. which gives insights into how the broader community of cloud native technologies is evolving as a whole.

If you're interested in sponsoring software engineering daily, we are currently looking for sponsors for Q3, we have around 24,000 daily listeners and if you're interested in getting your messaging to the ears of people who are interested in for example, cloud native technologies, this is the perfect podcast to get your messaging out there. Feel free to send me an email, jeff@softwareengineeringdaily.com. I'd love to hear from you. Now let's get on with this episode with Dan Williams.

[SPONSOR MESSAGE]

[0:01:50.9] JM: In the information age, data is the new oil. Businesses need data and there's no better data than real time data, which is why Amazon web services built Amazon Kinesis: a powerful new way to collect, process and analyze streaming data so that you can get timely insights and react quickly to new information. Here is the thing, websites, mobile apps, IOT sensors and the like can generate a colossal amount of streaming data. Sometimes terabytes an hour. That, if processed in real time, can help you learn about what your customers, application and products are doing right now and respond right away.

Amazon Kinesis from AWS lets you do that easily and at a low cost. With just a few clicks, you can start sending data from hundreds of thousands of data sources simultaneously. Loading in real time, it lets you process and analyze the data and take actions promptly. All you need to know is SQL. Kinesis also gives you the ability to build your own custom applications using popular stream processing frameworks of your choice and with Kinesis, you only pay for the resources that you use. There are no minimums, no upfront commitments.

To learn more about kinesis, just go to kinesis.aws and let's get streaming.

[INTERVIEW]

[0:03:28.4] JM: Dan Williams is an engineer at Red Hat. Dan, welcome to Software Engineering Daily.

[0:03:31.8] DW: Thanks, great to be here.

[0:03:33.7] JM: Yeah, it's great to have you. So, today we're going to talk about the Container Networking Interface and to get us to that place, I want to first talk about some broader networking concepts and we'll work our way to container networking because I think for a lot of people, it's kind of a scary topic that sounds a little low level.

Let's just talk broadly about the networking interface between two physical machines, just two physical hosts and then we'll work our way to VM's and then to Docker containers.

[0:04:08.1] DW: Sure. I mean, the basic level that you typically have is two machines connected via something like ethernet and you know, you connect a cable between them and each one of those has a network card in them that is exposed by the kernel. You have assigned addresses, IP addresses to those interfaces and then more or less they can talk to each other. So that part's pretty simple. Where it gets really interesting and you know, I assume we'll touch on this more during the podcast is when you have machines that are physically separated by much wider distances where of course you have to do switches, routers, all those kinds of things.

[0:04:47.9] JM: Explain a little bit about that background, because I actually know nothing about switches and routers and physical infrastructure. This has all been pretty much abstracted for me from my computer science education. You know, I've only been kind of CS for like six or seven years and I find this sometimes where I talk to people that have been around for maybe like a little longer than that, like 10 years, they might have more understanding of those things. But maybe give a little explanation for the physical infrastructure and why that starts to matter over longer distances.

[0:05:19.1] DW: Right. Well, you obviously need two machines to talk to each other over those long distances but we can't use ethernet over those long distances necessarily. At that point you have to start using things like bridges and switches to kind of translate between the different network technologies the two machines that needs to talk to each other.

For example, from the house where I work, it's all ethernet to my Wi-Fi router or it's Wi-Fi to the Wi-Fi writer and then from there, it's a short hop to the fiber gateway and then from there, it's fiber to somewhere who knows where and then from there it's probably also fiber to wherever I'm going to. You know, if I'm trying to go to Google or something like that. But then of course you know, Google's data center, it turns back from fiber eventually to ethernet most likely before it hits one of their servers.

So you do need things to adapt to various networking technologies, that's why it's called the physical layer, the actual physical thing that connects two machines or two jumps or hops in the path between you and wherever you're trying to get to. After the physical layer of course, then you actually need to start talking about layer two, which is something like ethernet itself and ethernet has protocol. We're kind of talking about the OSI network stack right now. If you're familiar with that and many of your listeners might be familiar with that as well.

The OSI stack has seven different layers, the other one's we actually care about for the purposes of this podcast are layers one through three, which are the physical layer and then kind of see the, I'm trying to think what the actual name is. The ethernet layer, that's layer two.

Of course where it could be something like, you know, the fiber layer, the protocols were fiber instead of ethernet and then layer three is the IP layer and that's — the IP layer is what's most interesting for container networking at this point.

[0:07:12.9] JM: Why is that? What goes on at the IP layer that makes it interesting for container networking?

[0:07:18.3] DW: Containers are typically very distributed and whether that means even the same data center, have a couple of containers running on one machine and a couple of containers running on other machine somewhere else and – or you might even have what are called federated clusters, at least in the Kubernetes world. There's Kubernetes Federation, which means you could have, say, a data center on the east coast and a data center out in the west coast and you want to actually allow the containers running in that Kubernetes cluster to talk to each other more directly than routing over the public internet.

What you do at that point is you want it to kind of virtualize or abstract via internet protocol or IP setup between those two containers and that's where the layer three stuff comes in and you want to kind of make all of them seem like they can talk to each other directly, like they are more or less on the local network. Or you can do things like routing between those but you steel need to do some interesting setup there to be able to get different containers to talk to each other directly and there are a number of different solutions for that like I said, whether that's IP routing, whether that's kind of VPN tunnels between two different clusters or whether that's something like Calico which does BGP inside the cluster to program routers and tell them you know, exactly where to send the packets. All sorts of different solutions.

But that's why it's interesting and that's why people have created so many different solutions is because that's where all the interesting stuff happens. The other thing is that because the containers are often spread out physically on different machines, you don't know exactly what the physical infrastructure is between those machines. It could be, you know, ethernet, it could be fiber, it could be something else, it could be ,you know, in very different locations. So IP, if you let the cluster administrator or the system administrator, it's not that kind of stuff. Just setup a network that's IP capable. Then, on top of that, using IP, all the stuff that I've just been talking about and kind of forget about kind of the layer one and like two stuff underneath.

[0:09:27.4] JM: To connect some historical dots, to get us to a discussion of container networking, there was a period of time and this is true, still for many enterprises where the state of the art was to have virtual machines that were, all sitting on a hypervisor somewhere on a physical host, multiple virtual machines and those virtual machines were networked together and you know, the more – I think the more trendy discussion or contemporary discussion would be "networked containers". Either on top of those VM's or just containers sitting across physical infrastructure. Let's just talk about the VM's for a second. What changed in the networking technology stack when virtual machines were getting popular?

[0:10:20.8] DW: Right. A lot actually changed because you know, at that point, the virtual machines are sort of virtualizing things, abstracting them again. You know, now obviously, if you're running multiple virtual machines on the same host, you need a way to make those virtual machines talk to each other and at that point, you're obviously not, you know, somehow, connecting physical cables inside that machine to different pieces so the VM's can talk.

You're kind of setting up virtual networking between those VM's. That opens up huge possibilities because all of that is – can be programmatically done through Linux Bridges or even something like SDN and you know, what was interesting there is now you need a solution to connect those multiple VM's or let's just say two VM's together and that can be through something like Linux bridges, that can be through, you know, either something like a SR-IOV where you, which is I think single root I/O virtualization, is that acronym and small tangent on that, SR-IOV is a feature of higher end network interface cards where they actually virtualize themselves in firmware and hardware.

They present themselves as you know, one to 64 different network cards and you can – they're actually at the PCI level and so you can take these PCI devices and put them into the virtual machine and then it just looks like you know, regular network interface card in the virtual machine but you get a direct access from inside the virtual machine back to the network interface card and then you can kind of, you know, more quickly or more performantly have VM's talk to each other. So that's kind of another solution there. But there are all these different solutions that people kind of created and can create because all this is virtualized. It's just so much more flexible than asking somebody to go into the data center and plug cables in to each other.

[0:12:12.3] JM: Yeah, so what's the best way to bridge this conversation about VM networking with container networking? Shat changes, I mean, it's almost like – kind of an orthogonal topic because you can have containers just running on that physical infrastructure without VM's involved, but it could also be connected because you can have a bunch of VM's on physical infrastructure and you could have a bunch of containers on each of those VM's. What's the best way to bridge these conceptual gaps?

[0:12:44.2] DW: You can kind of think of containers as, at least from a networking prospecting, as the same kind of logical unit as a VM. It doesn't really matter a lot of ways whether it's a VM or a container most of the time, people are just trying to run an application or a couple of applications. Obviously VM's are a little bit more flexible there in some ways because, you know, you start a VM and then you can run multiple things inside that VM.

You know, that complicates things a little bit, but at the network level for the host, the single machine that is running either VM's or containers or a mix of both. They basically can look almost exactly like each other. You connect the VM to the network on the host or you can connect one containers to the network on the host.

Containers kind of came around because there are some performance issues with VM's. You're obviously virtualizing an entire machine and so you have overhead because you have to emulate CPU, you have to emulate hardware. Emulate is the wrong word there but you have to sort of fake the infrastructure around those things so that the OS is running inside the VM thinks that it's running on an actual machine. That necessarily involves some overhead.

Whereas with containers, you get a lot of the isolation benefits of virtual machines while at the same time, you're still using the network stack of the kernel on that host. So you don't have any of that virtualization overhead. You're actually running direction on the network stack of the host. So that's one of the benefits of containers there is that you don't have that overhead and you can also do things more simply in a lot of ways because you don't have to deal with some of the virtual machine technologies.

You can setup the networking on the node or on the host itself and you don't have to setup networking inside the virtual machine again. So that's one way in which container's a little bit simpler and a little bit more performant in some cases. Now, that's not to say that virtual machines are necessarily slower because things like SR-IOV were developed to solve some of that problem from the networking perspective so that VM's could directly talk to the hardware on the node.

[SPONSOR MESSAGE]

[INTERVIEW CONTINUED]

[0:16:13.5] JM: When we start to talk about these container networking issues, is networking between two containers on the same host, is that going to be the same kind of conversation as networking between two containers on different hosts or what would be the difference there?

If I've got a container in one datacenter and then I've got a container in another datacenter that's across the world, how does that differ from the networking between two containers that are just sitting like super close on the same conceptual box?

[0:16:51.9] DW: I mean, obviously you're going to – if they're on the same box, you're going to have much lower latency, you're going to have fewer packet copies and you're probably going to have much faster access and that's the same with a VM as well because the packets can just go right through to the other destination.

When you jump between different machines then you have to deal with, you know, other users on that machine, going out through the same link. So you might not have full link bandwidth going out from the node to the switch, and congestion and other things like that. Whether it's a VM or a container, once you step outside of the host, it becomes quite a bit more complicated and heavily dependent on what the network infrastructure is that connects those host together.

You know, in the simplest case, with a container you would have two containers and they would have Linux Vive devices that connects those two containers to a common Linux bridge and so in that way, and you assign obviously IP addresses to those two containers that are in the same subnet and then those two containers can talk to each other directly.

Now, when you want to get that traffic outside of the node, you typically would take the network interface card that's on that node and add that to the bridge. Or, you would do something like network address translation from the bridge through the Linux Kernel, to that network interface cards that other applications that are not containerized can still access the outside world.

So it's fairly simple and it gets a lot more complicated if you want to do things like have overlay networks, which maybe we'll get to in a little bit. But then, once the traffic gets from the container through the Vive interface to the bridge and then to the network interface card and host, then it goes up to a switch and at that point it's mostly just IP routing although depending on how your setup is, it can involve also ethernet and things like ARP, address resolution protocol, and other stuff like that so that the two containers can actually – that container can actually find other containers out there.

Because most of the time with most of the orchestration systems, you're really just working at the IP level. Say you have container A on node A. And container A wants to talk to 10.5.6.7. Well, if 10.5.6.7 is the address of the container on some other node, you need to know what the ethernet address of that other node is, if you're on the same broadcast domain. When I say broadcast domain, that just basically means all of the machines are plugged into the same switch or the same series of switches so that they can talk to each other directly with ethernet.

In this example, the container A sends out an address resolution request to say, “Hey, anybody out there, if you have the IP address 10.5.6.7, tell me what your ethernet mac address is,” and that actually gets broadcast out to all machines in that broadcast domain, all the machines connected to that switch or that are on that V LAN. That’s obviously where things might break down if you have hundreds of thousands of machines that are all running containers. Then the container that has that IP address answers back, “This is my Mac address,” and then container A is able to send that data or that packet to that other container by addressing it directly with its ethernet address.

Now, if you have containers in different datacenters that are not connected via ethernet directly, you need to do IP routing or something else so that you can actually get that traffic between those two datacenters and that necessarily involves programming a router somehow to say okay, you know, everything that’ sin the 10.5/16 subnet is over on the west coast. If you ever encounter an IP packet that is 10.5.x.x, just send that to the router on the other coast. Then that router over there sees that and it kind of knows where all the rest of those containers would be in that datacenter and is able to direct that packet correctly. It can get pretty complicated.

[0:21:04.9] JM: yeah, to set some context, the reason we’re doing this show is because several years ago, Docker made containers easier to work with and so more people found out, “Oh containers make deployment a lot easier, they give you better economies of scale for your infrastructure,” and you know a series of dominos just fell after that and we started getting these exciting orchestration frameworks like Kubernetes. We had Cloud Foundry and Mesos at the time that were allowing people to build distributed systems that were easier to manage than the kind of wild west that came before, you know, having these well defined orchestration systems and then you know, after Kubernetes just really just lit a match in a room full of gasoline.

You know, the CNCF, the Cloud Native Computing Foundation was stood up to be basically the Linux foundation for container/cloud native infrastructure and bring some order to the chaos and to kind of help people be better aligned so that duplicate work wasn’t being done and maybe different orthogonal paths might be resolved in a way that was more amenable to everybody and have higher utility for everybody. So the CNCF stuff I’ve been reporting on a lot and one of the projects under the CNCF, which again is sort of like a Linux foundation for cloud native projects, it's a subset at the Linux foundation, one of these projects is the container network interface which you are working on. What is the Container Network Interface?

[0:22:48.7] DW: CNI is, at its most basic form, simply an API so that run times and command line tools and such can setup networking and it's mostly about containers. Obviously, it's got container name, though we're thinking of eventually expanding that towards things lie VM's and such, which maybe we'll get to later.

But I mean, at its base, it's an API and currently it's a simple kind of command based API and so there's currently version, which asks the plugin to return the version and there's kind of add and delete requests and the add request is from a run time to a CNI plugin and says, "Hey, I've got this container, please setup networking for it. Here are some details," and then the plug in will return to the run time, details like the IP address routes, DNS, those kinds of things.

That, in its most basic form, that's what it is. It's nothing more or nothing less, it's just an API. There are also some other components around CNI that make it easier to use and some – two of those are the configuration format so the specification for CNI also includes a config format, which is what the plug in consumes and how the run-time is supposed to format that information so that the plugin can understand it.

Currently, that's Jason, there's kind of the specification for CNI and it lists some examples for what the configuration could be. Then the other component – sorry, there's two more components actually. The other – one of the other components is LibCNI, which is a Go implementation of the CNI specification that a lot of projects vendor into themselves so that they can more easily use CNI in the plugins.

Last part is a number of reference plugins that the CNI project has developed so that people don't have to start from zero. They can kind of grab plugins that are interesting to them and composed them into a system that works for them. So that's the four parts basically; the specification, configuration format, LibCNI and the reference of plugins. Go ahead.

[0:24:59.6]JM: Could you talk more about what are the problems that the CNI solves?

[0:25:07.0] DW: Before CNI, a lot of run times, whether that's Kubernetes or Mesos, or you know, Docker, Rocket, or anything like that, they were all attempting to do this network setup themselves and CNI kind of grew out of the Rocket project as a way to encapsulate all of the network setup in a way that was more flexible and it proved useful for other projects and so that's one of the reasons that Kubernetes adopted it because Kubernetes itself wanted to get out of the business of trying to do networking.

Because that was not necessarily a core function of the orchestration. You know, yeah it is kind of a core function but there's so much variation and so much flexibility that people want out of container orchestration and container networking that Kubernetes didn't want to have to deal with all of that itself. It would rather make that the responsibility of network plugins and that also helps developer really healthy ecosystem because a lot of plugin vendors, because there's a standardized interface, they can just kind of slot in the stuff that they're working on.

That's why you have, I think at this time it's maybe 10 plus different CNI plugins. You know, I mean, that's official ones that are out there, released and that are like listed on the CNI website as examples. I've also heard of tons of, you know, like one-off proof of concept thing. It turns out that because it's an interface and because it's a very simple interface, it's very easy to prototype new ideas with CNI and so there's a lot of other projects and a lot of other people that, you know, I've run across that have written a quick CNI plugin to try something out to do what they want, or you know, just to kind of modify the behavior of Kubernetes in a way that they want.

The other cool thing with CNI is you can compose the plugins into kind of chains. So you don't need to write your own CNI plugin to do everything if you want to do kind of simple static local allocation, you can use the CNI host local reference plugin and then you don't need to deal with IP address management on the local machine if you don't want to. Or there's some plugin for DHCP. So if you actually want to put your containers directly on your network and grab DHCP addresses from them from your router, you can do that as well. You don't have to write that because that's already written for you and there's a well-defined way to call that.

So you can kind of build up the network setup that you want for your containers by using these difference kind of small CNI plugins to do specific tasks. I hesitate to call it the UniK's way of containers because that's a very contentious thing and it means a different thing to different people. But you know, sort of the philosophy of do one thing and do it well and a lot of CNI plugins do that. But then again, there's also the flexibility with CNI like I said, the composed plugins and so for example, the project that I work on, which just OpenShift, we actually consume a number of different CNI plugins and kind of build them up together to get the result that we want and we also do a bunch of other stuff. Open shift is kind of an example of fairly complex CNI plugin that itself uses CNI to call a couple of other CNI plugins to do very specific tasks.

[0:28:22.7] JM: Yeah, that's very helpful. I want to talk a little bit about OpenShift later, but let's zoom in on this concept of a CNI plugin. From what I understand, reading stuff and hearing what you just said, a CNI plugin is essentially the – anytime you want to build something that needs to be able to add a container to a network or delete a container from a network, that is a characterization of a CNI plugin.

So all CNI plugins need to be able to add a container to a network and delete a container from a network and my intuition on this is, because you're kind of trying to avoid zombie containers or memory leaks in your distributed system from just like these rogue containers that get created or spun up or they get partial failures and they just fall out of touch or I don't know? Maybe I'm wrong about that but tell me, I guess give some clarifying exposition on what a CNI plugin is meant to do.

[0:29:27.6] DW: You basically have it correct for the container lifecycle. When the run time says that it wants to start a container and that container needs network access, then it will run a given CNI plugin and say, "Hey, add this container to the network," and then of course, when that container is done or if that container has failed or whenever the container is no longer fulfilling its function, then the runtime will say, "Hey, remove this container from the network."

That necessarily involves some cleanups. For example, If you have allocated IP addresses that container, well we might need to release those IP addresses so it can be reused later for something else since IP addresses at least from an IPv4 are a pretty finite resource. Such the general life cycle is add and delete at some point later in time and the delete operation obviously as it cleans up post the resources that the container would use.

So that couldn't clear things like tearing down network interfaces, moving that container's network interface from bridge on the node. It could include updating routers to remove routes to that container. It's basically all up to the network plugin and CNI attempts to encapsulate all of those operations in the plugin itself, so at the run time doesn't really have to care that much about them. So the run time just knows that all it needs to do is add this container to a network or remove this container from the network and it keeps the runtime quite a bit simpler.

So for example with Kubernetes, again, there's no way that Kubernetes would want to deal with something like software defined networks and also have to deal with local networks – not local networks but host specific networks with just a simple Linux bridge all those kinds of things and so that's the great thing about having some kind of API in an abstraction like CNI, is that you can make that somebody else's problem but still benefit from it in a very standardized way.

[0:31:21.0] JM: Now was I right in my assumption that this is about preventing zombie containers?

[0:31:29.0] DW: Sort of. I mean a lot of that depends on the runtime itself and the runtime, whether that's Kubernetes or Rocket or anything else, will typically manage the lifetime of the container and so the runtime itself will handle the teardown. With that said, not to jump too far into it but that's tricky to get right especially when you have issues like I think what you eluded to was if the container fail set up in some way whether that's network related or not.

Then of course you need to detect that and if you have already allocated network resources to it, you need to tear this network resources down even though the container was actually never alive and there's also the complications around well what happens if your runtime crashes and gets restarted. Ideally you want to pick up where you left off because if the runtime crashes well, the container manager like Docker, Rocket or something like that probably hasn't crashed.

And so those containers are actually still running. You want the runtime then to list all the containers that are there and kind of re-read their IP addresses and state and just start from where it left off. But sometimes that's not possible and so you need to clean those things up. But yeah a lot of the container runtime problem is enforcing things like CPU limits, network limits, making sure that the container actually exits when it's done because often you are running applications in those containers that are necessarily trusted and so you need to make sure that the runtime, and to a large degree the network solution, enforces specific limits and make sure that the containers can't do stupid things so much as things.

[0:33:06.5] JM: I've done a few shows recently about service meshes and in my conversations about service meshes, I've asked both the guys I was interviewing like what are the bounds of the Kubernetes project and where – because the whole thing about the CNCF is Kubernetes is under this foundation but then also you've got a bunch of projects that are related to Kubernetes and just as an outsider who does not have programmed anything around Kubernetes. We just talked about it, it's not intuitive to me what is in the per view of Kubernetes and what kinds of things should be like stripped out of that project and put in a different initiative.

Can you talk a little bit about the boundaries of what kinds of initiatives belong in the Kubernetes project and why for example container networking is something that should be – like Kubernetes should not be thinking about container networking except to the extent that they should pick some project like the CNI they can outsource their networking interface to?

[0:34:19.6] DW: Yeah and just to be clear, I'm aware of some of the decisions that have been made but I am not one of the deciders of the direction of Kubernetes. So this is my impression of how things are going within the project. Basically and sometimes it seems like there is no bounds to what Kubernetes actually encompasses. There are so many things that are in the space and obviously that stuff group wake up one day and say, "Oh hey we need load balancers and service meshes," or something like that.

But over the past couple of years people have found those things to be useful and they get added. I think where, my personal opinion, where things should be more tightly integrated with Kubernetes is where there is a case to be made for management and statistic status, those kinds of things that cluster administrators and application developers would be interested in and would need to do their job. Obviously you don't want to get Kubernetes into the details of setting up Ethernet or something like that.

But at the same time as an application developer you definitely want to know, what's the IP address of my container? Is it healthy? Can I talk to it? Can I reach it on this port? What if I wanted to be reachable on a number of different ports from different places? Those are things that obviously intersect with the container orchestration and so thus are in the per view of Kubernetes I think and at least on the networking side it's more about there's so much flexibility in how people want to do networking that there's just no way that Kubernetes as a project, whether it's a project with the people that are involved in the project have the bandwidth to monitor, update, bug fix and watch, all the ways that you could do networking so that Kubernetes would natively do those things.

So I think where it makes sense, Kubernetes probably wants to spin things out and an example of that is probably Q proxy, which is the thing that by default does services with Kubernetes and that also grew organically in the Kubernetes project. But it turns out that a lot of people want different things from services and different things from the proxy and so there's been a number of different projects to basically rewrite the proxy and that's one component that I think people are looking at and seeing, "Hey there's actually other solutions for this proxy. Maybe we should also entertain those and try to make the interfaces that Kubernetes uses to talk to the proxy more generic and more well-defined so that other projects can re-implement it and experiment with it, do interesting things."

People have been working with IPVS for the proxy, there are cloud proxies. So where there's a lot of different ways to implement something where users of Kubernetes need a lot of different flexibility, I think it makes sense to move those things outside the Kubernetes project, whether that's an incubator under the Kubernetes umbrella but not necessarily part of the Kubernetes release process or core project itself. Or whether it's an entirely separate project like CNI. I think that's probably where the line should be drawn. It seems to me that's where increasingly the line does get drawn.

[SPONSOR MESSAGE]

[INTERVIEW CONTINUED]

[0:39:28.9] JM: You mentioned the importance of CNI plugins for building OpenShift. So OpenShift is a platform as a service built on top of Kubernetes. I think now is a good time to go into some details around why CNI plugins were or have been useful for building OpenShift. I guess maybe if you could give a little bit of the overview of the relationship between OpenShift and Kubernetes as well, which we have covered in some different episodes. There have been – we did a show with, was it Clayton Coleman? That's the guy who started OpenShift right?

[0:40:02.3] DW: Exactly.

[0:40:03.3] JM: Yeah, okay. Right yeah, so we have done some stuff about this, but yeah.

[0:40:06.9] DW: Well since you've had Clayton on then I don't need to go into any of the history as I understand it.

[0:40:11.2] JM: Well, but not everybody has heard that episode, I think.

[0:40:13.9] DW: Fair enough. I started working in OpenShift when I was getting ready for version three and I think now we're at version 3.6. So it's been a couple of years that I've been working on OpenShift and networking. But as I understand it, OpenShift version two, which I maybe thankfully was not around for was all based on Ruby and at a certain point they decided, "Hey, Kubernetes is already doing a lot of the stuff that we want to do. So why don't we move the project onto Kubernetes as it may take all the things that we were working on that Kubernetes didn't yet support and kind of port those to Kubernetes and because Red Hat is an open source company, all the things that Red Hat tries to do we try to do upstream or if we can't do it immediately upstream, we keep working to try to get them eventually upstream. That means that a lot of these components that OpenShift had developed outside of Kubernetes we keep working to try to make standardized Kubernetes specifications or implementations of those things.

So with OpenShift version three, OpenShift was based on Kubernetes and we've done a lot of work at Red Hat around storage, around the actual container runtimes, whether that's Docker, CRIO, other things and try to integrate those things into Kubernetes so that we can consume them again and it would shift and when I started with OpenShift networking, there was an OpenShift STN plugin that did not do anything CNI related.

It was an exact plugin, which is a previous plugin interface that sort of mirrored CNI but it was basically Kubernetes would just call out to a binary somewhere on the file system and say, "Run, add this network and then tell me the results," and so OpenShift used that and we want a little bit more flexibility in Kubernetes and so a couple of us on the OpenShift team tried to take a look and see how CNI would integrate and it turned out it worked out pretty well for Kubernetes.

So we added CNI support to Kubernetes and then we have moved the OpenShift STN project over to be an actual CNI plugin as opposed to be its own custom stuff that was kind of tightly integrated into OpenShift and Kubernetes. So at this point, OpenShift is a CNI plugin that you can't – OpenShift STN is a CNI plugin. You can't run it with pure Kubernetes yet because it does some OpenShift specific stuff but hopefully in the future we can remove that or at least make Kubernetes a little bit more capable so that we can handle the same lose cases in Kubernetes.

OpenShift. I think like we talked about before, they actually consumed a couple of other CNI plugins and what I meant by that is I mean that when OpenShift STN has asked to start a container network, add a container to the network. We do a bunch of stuff and then we actually called the host local plugin because we do use – we assign a subnet per node and so each container on that node gets an IP address in that subnet and the host local plugin does exactly that. It's kind of a static IP address allocation plugin on that particular note.

So OpenShift calls us to host. Those local says, "Give me an IP address free from the pool for this container," and then OpenShift consumes the results, does some other stuff and then passes that back to Kubernetes and that allowed us to remove a whole ton of our own IP address allocation code from OpenShift STN because guess what? CNI already providing that and then there's a number of other CNI plugins that as we look to the future that will probably consume in the same way as well and one of the things that I've tried to focus on as well is once I find something that we need OpenShift to do if it can be a CNI plugin, then I will actually write a CNI plugin and submit that to the CNI project to do that thing and then eventually consume that using OpenShift STN in the same way that we consume host local and a couple of others.

[0:44:15.9] JM: For people who are still confused about what the idea of a CNI plugin is, what are some examples of CNI plugins that you think people might want to build in the future?

[0:44:28.3] DW: That is a great question. One of the things that we've seen a lot of interest in is plugins for Linux Kernel device types. Like VLAN or IP VLAN, MAC VLAN. I am trying to think of some other processor IOV even – I don't mean again at its base, all of these plugins are doing is you give the plugin a container and you say, "Hey plugin, put this container on this network with some configuration that I am giving." So for example on a VLAN plugin, what that VLAN plugin would do would be to create a VLAN interface off of your ETH0 or something like that.

And then using Linux network namespaces, you move that VLAN at your phase into the plugin – or excuse me, into the container. So at data point the container actually is on that VLAN that is seen from a switch outside the host. That's one example. There are also plugins for, like I said I think DHCP. So if you want a container to get it's IP address from DHCP and that's actually a tangent we could go down is IP address management plugins.

There's two plugin types for CNI. The first one is the main plugin and that actually sets up the interfaces and such but then the IP address allocation is completely different because once you've set up interfaces, there's many different ways that you could decide what IP address the container gets and so, CNI provides a second kind of plugin and API's to talk to that plugin that are specifically about figuring out what IP address to get for a container.

But you know at its base, CNI plugins are just small binary executables that get run by the runtime and those executables are past in action, some configuration on a couple of other properties. They do their job, they exit and before they exit they print to standard out their return result, which is also kind of an adjacent structure that describes what interfaces they've added to the plugin and what interfaces they've created on the host, what IP address they have decided the container and maybe what IP addresses have been assigned to with interfaces on the host by the plugin, and then routes, DNS, those kinds of things and then the runtime consumes that, in some way at least in the Kubernetes case that means saving the IP address and putting that into the Kubernetes API so that as in the app developer or cluster administrator you can actually find out what IP address your container is using.

[0:47:00.5] JM: So you know one thing I am curious about, this is sort of an orthogonal question, you know I have done a lot of shows about containers and Kubernetes and these related topics. One thing I don't understand is, what's the difference between a Rocket container and a Docker container? Are these differing factions or are they different used cases for different types of containers?

[0:47:29.6] DW: I think pretty simply you can think of them as different factions. I mean obviously everybody knows Docker but there's also Rocket, there's also CRIO, which is a new project that is lighter weight than Docker but basically does the same thing and the thing that all these do is actually run a particular process in a containerized way and so the base of containers or the thing that containers are built from Linux or namespaces and there's a couple of different namespaces.

When we say "namespaces" that just means a way to isolate a particular set of properties from all other things or all other of those properties on the system. So some of those name spaces are the network name space, the process name space, the file system name space and those are the most interesting ones at this point. So for the network name space, it's basically just each container, each thing in that network name space gets its own network stack.

So you can think of it as multiple copies of the network stack running in the same Linux kernel. Or multiple copies of the file system view running in the Linux kernel or multiple copies of the process view running a Linux kernel. So when you create a new network name space, the application that is running in that network name space can't see anything. It has no interfaces except for the loop back interface but nothing else.

So it's effectively isolated network bias and anything you do in that network name space has absolutely no effect on any of the other network name spaces, whether those are in other containers or whether that's the host and the same thing goes for the file system in process namespaces. You know when you create a new process namespace, the process running in that new process namespaces can't see any other processes running on the system.

And so using these kind of building blocks, you can build up what we think of as Linux containers and there are many different ways to do that. I think even before Docker there was LXC, if I'm remembering correctly, that was a way to use some of these facilities. Even before that, I'm trying to remember what that was called. I worked on –

[0:49:44.5] JM: Zones.

[0:49:45.4] DW: Zones, yeah there was Zones but there was also, I worked on the OLPC project 10 years ago and they were investigating a solution that starts with a V that basically did the same thing. Sandboxed processes, I don't remember what that was but anyway, I mean it's obviously a concept that has been around for a very long time and it turned out that once Linux got namespaces that was the thing that clicked and that was what allowed all these different solutions to come out.

And Docker happened to be the one that got the most traction and I think still has the most mindshare or maybe we should call them [inaudible 0:50:20.6], who knows? But Docker basically, it does the most of the same things that Rocket does. They both are, their core function is isolating processes, whether that's for isolation from all the different kinds of name spaces. So it's a fully isolated container or whether it's a container that just uses kind of the file system and software distribution aspects of containers but not so much the network isolation aspects of the process, isolation aspects but that's probably for a whole different podcast.

[0:50:53.0] JM: Okay, tell me if I have this idea wrong, but one way I see the project in the CNCF is you've got these projects that have some significant overlapping functionality. Like Kubernetes and Mesos and maybe you have these other projects like the two service meshed projects that I did shows on, Istio and Linkerd. I don't think Istio is in the CNCF yet, but it's in this space and so you got these projects and then you've got Rocket and Docker and these projects were that's basically some overlapping, people stepping on each other's toes and I know this is a big positive sum growing market. Everybody is going to succeed, it's totally fine but nonetheless there could arise conflicts between those projects.

But then you've also got these projects that are fairly innocent. It seems like the CNI, this is something that everybody wants. It seems like it doesn't really harm anybody to come to a conclusion on what the CNI should be. Am I painting a picture? Is that an accurate picture? Am I injecting a narrative that is not really accurate?

[0:52:04.5] DW: I think all of it is just kind of a reflection of in some ways the beauty of open source. Yes, it's always been this way. Everybody, since the beginning of open source more or less, has created similar projects with more of us and more aims and you know in some ways that's why things work because people can create different things. They are not necessarily constrained by one particular project. They can fork things, try out new ideas and then in the end, the best of those ideas end up getting brought back into projects.

I feel like once – if you are in an area where it's really new, there's a lot of experimentation, this is just the natural way it works and then once people have figured out how things worked and what the general way things happen is, people kind of consolidate around one or two particular projects. Some of the other ones may live on, you can think of Linux or FreeBSD and all different BSD's as well. Some of them are more popular than others but you can argue that Linux is one.

People kids of consolidated around Linux or with distributions, there's many different distributions. So they all fulfill different requirements and niches and I feel like that's the same thing with a lot of these different projects in contain land right now because containers are so new and it's such an interesting area to work in, everybody has different ideas and eventually the best of those ideas will probably come back and become integrated into the projects that get the most mind share with the most users.

But that of course doesn’t prevent somebody else with coming up with another project that has multi cool ideas in it as well even after people have started using, for the most part, one project. So I mean I personally don’t see a huge problem with having a bunch of different projects that they all do the same thing. That’s the natural way of open source. It obviously can run into some political issues and their personality issues with different personalities who run these projects and there’s always some “not invented here” type things but in my experience, that always works itself at the end and I think the committee is better for it.

[0:54:13.5] JM: All right Dan, well I want to be respectful of your time. To close off, do you want to give a preview for what’s coming down the pike in the CNI project and what you're focused on right now?

[0:54:24.5] DW: Definitely. One of the things that we recently added was multiple interfaces and ordered imn a couple IP addresses and so it’s a lot more flexible when plugins can return and we’ve used that or kind of piggy backed on that to do kind of the plugin composition stuff and we call those Conf Lists or configuration lists.

So you can kind of build up a chain of plugins and do different things and then of course the final result gets to return to the runtime. This is something that we – on the Kubernetes side, we intend to use in Kubernetes to allow IPV6 support which Kubernetes does not really currently support. Obviously for that, you want to have multiple IP addresses, not just because you’ve got dual stacked V4 and V6 but also because with V6, you always almost – almost always have both the link local address and you know, global site, local address.

That’s one of the big things and the other one is, we’re going to probably have a release of CNI in the next couple of weeks again and we’ve done a whole bunch of cleanup for IPV6, we’ve added a bunch of – not a bunch but quite a few other plugins for example, VLan plugin, looking at some IPV6 SLAAC plugins, not Slack as in the chat communication service, but the SLAAC which is the way that IPV6 addressing usually works.

We're also exploring ways to make CNI a little bit more dynamic. Currently the commands are just version add and delete fairly simple but there are some use cases where we'd like to be able to ask the plugin. Hey, is everything working correctly and if it's not, be able to communicate that back to the run time so the run time can do something intelligent. Whether that's tear the container down or you know, maybe remove that network and we add it, that kind of thing.

With CNI, a lot of the to do list and the for direction is obviously driven by the consumers because those are the ones who know what they want a CNI to do. For the most part right now, that's Kubernetes, that's also rocket and core OS, make some other community members as well. I mean, the other thing I do is, you know, put a plug out for CNI, if anybody's interested in this phase, please get in touch, we have mailing lists, we have slack channels, we have IRC channels and then obviously GitHub as well.

Always looking for new contributors. You know, we've had in the past had a pretty large backlog of pull requests and issues but we've been working diligently to try to become more responsive on those things and to work those things down. You know, if you're interested, great, be involved, we'd love to have you contribute code or contribute testing or documentation or anything at all to the CNI project. If you have suggestions for the CNI project but want to contribute code, that's great too. Anybody and everybody obviously could use help and you know, could use all the suggestions. It's obviously an open project as well on GitHub and elsewhere.

[0:57:24.8] JM: All right Dan, well I appreciate you coming on the show, it's been great talking.

[0:57:28.0] DW: Yes, thank you. Thanks for the opportunity.

[END OF INTERVIEW]

[0:57:33.8] JM: Simplify continues delivery. With GoCD, the on premise, open source, continuous delivery tool by ThoughtWorks. With GoCD, you can easily model complex deployment workflows using pipelines and visualize them end to end with the value stream map. You get complete visibility into and control over your company's deployments.

At gocdy.org/sedaily, find out how to bring continuous delivery to your teams. Say goodbye to deployment panic and hello to consistent predictable deliveries. Visit gocd.org/sedaily to learn more about GoCD. Commercial support and enterprise add-ons including disaster recovery are available.

Thanks to GoCD for being a continued sponsor of Software Engineering Daily.

[END]