Intel

Cloud Native Packet Processing on Kubernetes with the Cloud Native Data Plane

CORPORATE PARTICIPANTS

Xiaojun (Shawn) Li
Intel – Sales Director, Next Wave OEM & eODM

Jeff Shaw
Intel – Cloud Software Architect

PRESENTATION

Xiaojun Li

Welcome everyone to the Intel Network Builders Insights Series. I'm Shawn Li, Sales Director, Next Wave OEM and eODM. Next Wave Communications is a sales organization at Intel, and I'm your host for today's webinar.

Thank you for taking the time to join us today for our webinar titled:

“Cloud Native Packet Processing on Kubernetes with the Cloud Native Data Plane”.

Before we get started, I want to point out some of the features of the BrightTALK tool that may improve your experience. There is a Questions tab below your viewer. I encourage our live audience to please ask questions at any time. Our presenters will hold answering them until the end of the presentation. Below your viewer screen, you will also find the Attachments tab with additional documents and reference materials, which pertain to this presentation. Finally, at the end of the presentation, please take the time to provide feedback, using the Rating tab. We value your thoughts and will use the information to improve our future webinars.

Intel Network Builders Insights Series takes place live every month. So, please check the channel to see what's coming and access our growing library for recorded content. In addition to the resources you see here, we also offer comprehensive NFV and 5G training programs through the Intel Network Builders University. You can find the link to this program in the Attachments tab as well as a link to the Intel Network Builders Newsletter.

Today, we are pleased to welcome Jeff Shaw. Jeff Shaw is a Cloud Software Architect in the Network Platforms Group at Intel. He develops packet processing technology deployed by cloud service providers and the comm service providers. He wrote the first optimized data path for the Intel network drivers in DPDK. Since then, he worked on kernel and userspace driver network stacks, crypto offloads, switches, and SmartNICs.

Welcome, Jeff, and thank you for taking the time to join us today. I will hand it over to you to start off. Thank you, Jeff.

Jeff Shaw

Thank you. Thanks, Shawn. Thanks to the entire Network Builders team for inviting me to present for the Insights Series. I know I've personally learned a lot from the Network Builders courses and a lot of these webinars. I've watched them before. So, it's really a pleasure to be here.

So, first, obligatory disclaimers. Make sure everybody reads and understands these and to learn more, you can go to the Intel site if you want.

Let's start off by defining what I mean when I say, “Cloud Native”. You'll find a lot of definitions out there, but I think this one from the CNCF describes it pretty well. Cloud Native is all about deploying scalable applications in all kinds of different environments. This is normally done by deploying applications in small pieces called Microservices, which are all managed and observed by a container orchestration system like Kubernetes. And these microservices are deployed, scaled automatically, depending on configuration, specified using some declarative API like the Kubernetes API. Great, so we have a definition. Let's unpack what all that means.

First, applications are being deployed in all kinds of different environments, this includes public, private, hybrid clouds. We're seeing businesses deploy parts of their infrastructure in traditional on-prem data center environments, so they can leverage existing capital expenditures, software investments, any other reasons why somebody might run on an on-prem environment, while other parts of the infrastructure are getting deployed on public cloud to gain scalability or be positioned geographically closer to its users at the network edge, which of course will then improve the end user's experience. But to achieve optimal operational efficiency, businesses need to span both private and public clouds, thus creating this hybrid cloud environment.

The second piece of the puzzle requires microservices to be deployed on consistent, relatively common infrastructure. The Cloud Native apps can't be developed as a single monolithic binary expected to be deployed at scale. They are more and more often composed as microservices, so each individual piece can be continuously improved and deployed with greater flexibility. And to be able to scale differently depending on quality of service, cost, latency, space, physical space constraints, all kinds of different things.

To achieve the economy of scale for compute resources, literally thousands of nearly identical servers have to be installed in exactly the same way across all different kinds of environments if you want to minimize cost and simplify deployment for your end users.

And finally, Cloud Native enables manageable, observable, and automatable infrastructure that has to change frequently. Many businesses that have embraced Cloud Native have thousands of services in production, if not hundreds... and deploy hundreds, if not thousands of times a day. So, obviously, this requires robust automation in order to continuously deliver these services with minimal or zero downtime. And observability is key here. So, when something goes wrong, you can't just start SSH-ing into a thousand different machines, reading all their logs, that's just simply not going to work well. So, you need manageable, observable, automatable infrastructure. So, that's what it means to me to be Cloud Native. So, let's see what this means for our network transformation story.

So, going back to the last decade or so, SDN, NFV have transformed the way network applications and network infrastructure are deployed. The result is this disaggregation of hardware/software. So, now applications can run virtually anywhere. And the next wave that we see coming is the Cloud Native transformation. So, platforms built on Cloud Native principles make it easier to introduce new services, improve resource utilization, ultimately provide a better experience for end users. So, as the hyperscalers have shown us, Cloud Native delivers the TCO benefits and agility required by businesses navigating this network transformation, as long as emphasis is placed on a few key areas that we still have some more work to do.

First, software has to be abstracted from the underlying hardware, to be able to flexibly deploy application components anywhere in the cloud. This is pretty much exactly what SDN has been driving for over a decade. So, there's nothing new here for network infrastructure.

Second, the scale offered by Cloud Native is simply not manageable without automation. It's not feasible to roll a technician out to thousands of nodes all over the world to install some new service manually. It's just impossible to do that at the scale that's required.

And finally, we need to embrace this idea of microservices. So, I know this is challenging. And for network infrastructures, a lot of the stuff is pretty tightly coupled. It has all kinds of different performance requirements, latency requirements, a lot of it is run to completion. So, it is harder to do for network infrastructure, but a prime example of this is the design of the 5G core network, which sets out to separate the data plane/control plane functionality and modularize the design of its components. So, you can independently scale and have a flexible deployment wherever possible. So, it's not easy. But as you can see, the Cloud Native transformation has already begun.

All right, let's go next slide here. All right. So, embracing an application development approach that uses the Cloud Native computing delivery models fuels innovation, increases velocity, though the approach does come with its own challenges. Take microservices, for example. When you have a monolithic application, you break it down into a bunch of distinct services all decoupled from one another, you remove the ability for the services to communicate with each other directly through traditional means, like in-process queues or shared memory. So, for example, when a network packet is handled by one part of the application, it can't just call the function in another part of the process to perform the next set of operations on the packet, because the next part of the pipeline is implemented in some other service that's running maybe locally on the same host or somewhere else in the network. So, you have to connect these services through the network using communication like RPCs, or some other well-known API over HTTP or something like that. And of course, this increases demand on the network, especially dozens of services deployed simultaneously across the same nodes. These RPCs tend to be pretty small packets, small messages, small objects that get transferred all over the place. So, we see a lot of small packet performance and the infrastructure is continually-- continues to be required.

So, service meshes help to set up, optimize, observe all the network traffic, but you still require significant compute power to handle all of this data. So, this is one area we hope to address with CNDP, but I will talk more about that later. I just want to point out that using the Cloud Native agile approach to network functions can improve our ability to deliver new and innovative functionality.

So, next, I want to introduce a couple of concepts, which are fundamental to the deployment and functionality offered by CNDP. First one I want to introduce is Kubernetes. I'm sure most of the people watching this talk are already very familiar with Kubernetes. So, I won't go into too much detail. But if there are any new people, I'll give a brief primer here. There's plenty of info on videos, tutorials that go over all aspects of Kubernetes. I know I've watched them. I've watched a lot of them. So, go check those out.

So, what is Kubernetes? Simply put, it's a system for managing containers. Typical implementations consist of some computers running a control plane shown here on the left side of the diagram. Control plane machines manage many, many more computers that run containers, these are called Worker Nodes, that is shown at the right side of the diagram. The collection of the worker nodes is called the Cluster. Each one of the control plane nodes run at least four things, which I've tried to show here. These aren't the real names of those things. That's just my distilled version of them. But basically, what the control plane has to do is store information about the cluster, schedule containers on nodes, control how the nodes are managed. So, if a new node gets added to a cluster, it needs to know about that. All of that is coordinated through a well-defined and versioned API. The worker nodes, they run at least two things. Kubelet is responsible for running containers, while kube-proxy is responsible for networking, either using the host native network stack or managing connectivity on its own. And I know this is a very simplified view of how things work. But this is really just an introduction to frame the discussion.

The unit that gets scheduled on a worker node is called a Pod that can consist of one or more containers. Most of these pods run a single container, but it's common to pair a primary container with a secondary container. And the secondary container is called a Sidecar whose job it is to do related tasks that aren't really part of the primary container. So, for example, if you have a CNDP application that is doing some kind of network function, and it exposes metrics through a Unix socket that are consumed and served by another container, by a Prometheus client, for example, the CNDP application implementing the network function would be the primary container, while the Prometheus client is the sidecar. So, keep in mind all these terms are defined in the Kubernetes docs. So, this is, again, just a high-level summary.

A couple other terms that I want to mention that are going to come up later are CNI, which is the Container Network Interface, as well as device plugins. CNI is responsible for implementing the Kubernetes network model so that pods can communicate with each other. You can't run a cluster without a CNI. The device plugin provides pods access to hardware, which requires special initialization and configuration. For example, CNDP applications running in containers that require least privilege can use an AF_XDP device plugin to program the XSK map for a device, which leads us to the next topic that I want to introduce which is AF_XDP.

So, what is AF_XDP? AF_XDP is a new socket address family, which hence the AF in AF_XDP, specifically created for high-performance packet processing. As long as the underlying driver supports it, AF_XDP provides zero-copy interface between a network interface and userspace application processing the packets. It is different from conventional TCP/UDP, IP, raw socket since the kernel stack is actually bypassed here. So, you can see the data path for AF_XDP, and zero-copy is shown on the left side where packets are processed by the device driver. And then they land directly into userspace via that AF_XDP socket. And as long as you map the memory between the userspace application here and the kernel, then packets are DMA-ed directly from the device into that userspace memory.

The feature was first introduced-- actually, AF_XDP was first introduced four years ago, which some people are surprised that this has been around for so long. It was actually, I think, yesterday-- four years ago as of yesterday that the commit got into the kernel. And in addition to performance improvement coming from the batch processing in AF_XDP, eliminating these packet copies, as well as the low overhead memory buffer management benefits that you get from XDP, there's operational benefits as well. So, for Cloud Native apps that need to operate in different environments, having an interface like AF_XDP that's built into the kernel provides a lot of flexibility. So, even if the underlying network driver doesn't support all the features of AF_XDP, for example, like zero-copy, the application can still run, so you can still open an AF_XDP socket on that particular device, and the kernel just implements AF_XDP for you, so you don't get the zero-copy, but you can still run your applications anywhere.

Last thing to mention on that is the XDP part of the name, it stands for Express Data Path. It's a subsystem that's used to redirect the ingress packets to userspace. To do this, it uses an eBPF program that's installed in the kernel. That instructs the network driver to redirect packets to the AF_XDP socket, which we call an XSK. And CNDP provides APIs to abstract AF_XDP sockets, which is part of the reason why I'm talking about it.

And one more thing to point out. I didn't really go into detail on eBPF or XDP. Those are described and discussed at length in a lot of different presentations. So, that's not what this one's about. I just wanted to talk about AF_XDP, because it’s pretty fundamental to how CNDP works.

So, those are the two background topics I wanted to discuss. Kubernetes, AF_XDP. Now, I'd like to introduce Cloud Native Data Plane or CNDP.

So, first off, we're very excited about this new project, you can check out the website cndp.io. Also, take a look at the code on GitHub. It's a project that's been developed for the last couple of years at Intel, using all of the best practices that we've learned over the past decade long or more in doing this packet processing stuff. And we've only, as of a couple of weeks ago, made this project open-source, public, and open-source, so we're very excited about it.

First, why do we need CNDP? Why am I talking to you today? Well, historically, it's been difficult to orchestrate packet processing applications. So, they generally require advanced network capabilities, specialized network hardware, they often come with stringent throughput latency requirements, or all of the above. But as I mentioned before, businesses, as they realized the benefits of Cloud Native, we're seeing hybrid and multi-cloud deployments becoming a lot more prevalent. In the hybrid approach, businesses take advantage of their existing on-premise enterprise data center, but they still have the elasticity to extend deployment to public cloud to achieve cost and scale benefits. We need Cloud Native Data Plane to meet the composability, automatability, scalability, performance requirements of this new network infrastructure. So, CNDP addresses this gap by providing a lightweight packet processing framework designed and built specifically for Cloud Native applications.

So, what is it? It's a purpose-built data plane. It provides a framework for packet processing microservices that's closely aligned with Kubernetes. It builds on top of AF_XDP technology that lets applications run virtually anywhere without the knowledge of the underlying network hardware. Using the AF_XDP device plugin and CNI, it's possible to provision, orchestrate, and manage the data plane using Cloud Native practices.

So, as comm service providers, enterprises look to deploy their applications in the cloud and at the edge, workloads may not seamlessly migrate due to various hardware incompatibilities, or for other reasons, CNDP tries to address these challenges. So, applications like firewalls, gateways, load balancers, 5G UPF et cetera can benefit from CNDP’s optimized packet processing libraries to implement the normal stuff like packet classifiers, flow tables, ACLs, routing on top of network interfaces that are using AF_XDP.

So, the application configuration. CNDP application configuration is applied through automation-friendly JSON. And we have built-in telemetry and metrics that are exported-- that can be exported through a Kubernetes sidecar to Prometheus for easy cluster level management, monitoring. And additionally, this is something that we've added recently, and we're working on continually, we provide language bindings for Go and Rust. So, that gives developers a lot of flexibility to create their apps on top of the Cloud Native Data Plane. As more applications get composed of microservices, the communication that happened in shared memory is now going to be handled by a service mesh. So, we see a lot of need to accelerate this, and Intel is doing a lot of work in this area. We are looking--

CNDP has its own network stack, which we think, in the future, still currently a work in progress-- we have UDP/IPv4, we're working on TCP/IPv6. But we think in the future, we can use this to accelerate service mesh, so we can free up cycles to run even more services. TCP makes it easy to automate and orchestrate packet processing applications onto Cloud Native platform. And it's an open-source community-driven project, visit us at cndp.io. That's my pitch.

Let's talk about the library. So, we'll go into a little bit more detail on the libraries, then I'll talk about the stack. And then I'll finish up talking about our Kubernetes deployment model.

So, first, libraries. So, CNDP, it's not an application by itself, though, we provide sample applications. But really, it's just a collection of libraries that help developers build their applications. So, we have core libraries, shown in the top left here. These core libraries depend on one another, or they're developed hierarchically. So, we have these low-level libraries, like logging, OS abstraction, CNEs, just the... our version of the environment.

And then we have these application libraries. And these application libraries build on top of the core libraries. They don't have a lot of interdependencies. So, you can mix and match these however you want. You don't have to use them all. But this allows developers to package the libraries that they want to in their application without buying into the whole framework wholesale.

Core libraries offer the typical functionality you'd expect from packet processing framework, libraries for logging, OS abstraction, basic runtime, that sort of thing. On top of these core libraries, we have the buffer management APIs. These are mostly ported and stripped down from DPDKs, mempool, Ring, and mbuf. And when I say ported, I mean copy/pasted, and just basically used wholesale. This is the most optimized implementation out there that exists. So, we definitely sample wherever we can to reuse what we can.

One thing that we changed, so since physically contiguous memory is not required for AF_XDP packet buffers, we removed that emphasis on HugePage-backed memory. So, we were able to get rid of a lot of code there. We just use Nmap Syscall, which you could still use to allocate memory from HugePages. So, you can still get those performance benefits without having to pre-reserve memory all upfront, and scan for HugePages, and map them, and sequence them into contiguous pages, that sort of thing.

On top of that, we have two different APIs that abstract network interfaces. You can use a low-level XSK Dev API, which is just a wrapper around Libbpf now Libxdp. So, you can manage your AF_XDP sockets. You can use the CNDP's packet mbuf allocation scheme, or you can bring your own, so you can integrate with-- if you already have your own application that has its own buffer management et cetera-- you can integrate that directly without needing to change the code.

We have a higher-level abstraction that's called Packet Dev, it has a pluggable driver interface. We currently have drivers for AF_XDP, af_packet, memif, we have Ring, which is like DPDK's Ring. And we just added a TUN/TAP driver as well.

What else? So, just a quick note on the packet processing libraries. They tend to do-- packet processing apps tend to do common tasks like packet classification and filtering, managing timers, routing. So, we provide libraries to help with that. Again, most of these are just directly ported from DPDK. We have the Rust/Go language bindings, and we have a Docker file and a Kubernetes deployment spec as well, which you can see that how we build our example container, we deploy in a pod with our Prometheus sidecar.

The last part of the documentation, obviously, and the public APIs documentation generated with Doxygen, user guides are generated with Sphinx, you can check out the latest docs on the website for the 22.04 release.

Of particular interest, one area that I'm particularly excited about is the network stack that comes with CNDP, which is CNET, it stands for CNDP Network Stack, currently supports IPv4/UDP, actively working on TCP/IPv6. You can actually see that development happening right now. It's happening on GitHub. So, it provides sockets-like interface called Channels, which is zero-copy interface, lets application developers listen on specific ports, register callbacks, handle packets for given connections, that sort of thing. It uses familiar sockets, like APIs, like Sync2. Configuration, use your Linux-- standard Linux tools. Remember, we're using AF_XDP here, so we're just opening a socket on an existing Netdev. So, if you configure IP addresses, routes, whatever on that particular Netdev, that information gets reflected to the CNDP application through Netlink. So, you don't have to do any-- you don't have to learn any new configuration tools or do anything extra. We're just taking advantage of the existing infrastructure that's already there.

To demonstrate the functionality of the CNET stack, we provide an example on integration of the quicly library. So, we have a QUIC server and client that just sits on top of the CNET stack.

Implementation details for the stack itself. So, it's written as a set of graph nodes. This should be a pretty familiar concept to a lot of people doing userspace packet processing these days. Packets pass through a directed graph. Each node processes multiple packets at a time. This allows the developers to extend the stack by adding new nodes for whatever desired functionality they want. So, you can have a new node if you wanted to for-- to do like encryption/decryption, to do encap/decap address translation, deep packet inspection, any kind of advanced flow classification, connection tracking et cetera. We don't have graph nodes for these, at least not yet. But the modular nature of the graph architecture makes it easy to plug these new graph nodes in, depending on whatever functionality is required.

The example graph on the right with the highlighted nodes, it just shows the nodes that get traversed through a locally destined UDP packet. So, first, those packets come in through the ethernet interface on top, they get classified. The ptype, which is packet type here, we determine that it's IPv4, goes to IPv4 input. This is not a remote packet. So, we handle it locally. Here, we look at the protocol, it's UDP here. We check to see if we have a protocol control buffer, or block for that thing. And we see if we have a socket open, and we call the callback that was registered by the application when it originally opened that socket.

So, I mentioned QUIC, but we expect we can use this stack, hopefully, once we get TCP and some more of the functionality implemented. We can use it to accelerate all kinds of other things like RPCs, and service mesh, and stuff like that. So, we'll see.

So, that was a brief intro on CNDP libraries and the CNET stack. Let's talk about how we deploy our CNDP applications on Kubernetes.

So, first, big shout out to the-- I meant to put a link here, so I can probably put a link in the attachments here, but the AF_XDP device plugin, the AF_XDP CNI, this is a separate repository on GitHub, and I'll provide the link. I don't want to accidentally tell you the wrong name here. But this is done by another team at Intel, and it's a lot of great work, and we are using it.

So, as part of the CNDP repo itself, we have the Docker file that you can build our reference container image and that package is the example CNDP application that does basic forwarding. And we also package a Prometheus client written in Go. We have the example pod spec that declares two containers. So, that's what's shown here. One is running the primary packet forwarding example application called the CNDP Container. The other is the sidecar which serves the statistics for the Prometheus client. The two containers communicate through a Unix domain socket. And then the sidecar runs that web server that serves the statistics over the Prometheus API, back to the Prometheus agent or whatever running on the cluster.

We use the AF_XDP device plugin to program the XSK map for the AF_XDP sockets opened by the application. Doing this allows the CNDP application to run without the normal privileges that are required like sysadmin capability, which normally is required to load the eBPF programs, configure some of the options on the AF_XDP socket. The AF_XDP CNI's job is to program ethtool filters, to configure RSS queues, flow director rules, and then move that device into the pods network namespace. But once the CNDP container comes up, it communicates with the AF_XDP device plugin over a Unix domain socket to program the XSK map, and any other options like Busy Poll that require privilege. But once that's done, the pod's ready to process packets.

Another thing to mention here is the AF_XDP devices are running on a secondary network. The pod still has its primary network for API communications, liveness probes, and the other networking that gets done through the native CNI. We're showing flannel here, but really the native CNI that's running doesn't really matter for the primary network in our case. Mostly it doesn't matter.

So, one thing that we're currently actively working on that's lacking in this existing design is that we have to pass that entire Netdev through to the pod. So, obviously, this isn't going to scale since the machine's definitely not going to have enough physical devices to pass through to each pod. So, we need a way to allocate specific queues from a given Netdev or a given physical function. A couple of ways to do that. First way is just to create virtual functions using SR-IOV, pass those virtual functions to each one of the pods. This is fine as long as that virtual driver supports AF_XDP and zero-copy mode, if you want to get all the benefits of AF_XDP, which not all drivers-- not all the virtual function drivers support that.

Another option that we're looking into is using the Linux Devlink system to create sub-functions for a given device. And then the sub-functions are passed through to the container. So, this allows applications to scale out much wider and make better use of network resources. So, look for those improvements coming in the weeks and months. It's a lot of moving parts involved on all the different Linux kernel mailing lists, as well as obviously our development here too. That's pretty much it.

I want to finish up and have time for questions. So, let me summarize a couple of the key points here.

Cloud Native uses cloud computing delivery models following up on SDN and NFV. We're ushering in a new era which adopts the principles of cloud computing and applies them to network infrastructure. So, we use Linux and Kubernetes technologies to balance abstraction and performance. Finally, CNDP is a framework for Cloud Native packet processing that helps developers realize these network infrastructure goals. And if you'd like to find out more, feel free to reach out to your Intel rep, or visit the cndp.io or come chat with us on CNDP GitHub.

And that's really all that I had.

Xiaojun Li

Thank you, Jeff. Thank you. And we have some questions for you. First question, "Why would you choose user pace networking when the kernel stacks work well already?"

Jeff Shaw

So, there's a couple of different reasons why you might choose userspace networking versus kernel networking. So, using a typical TCP-- AF_INET socket or something like that for TCP or UDP, that works great, as long as your application is speaking TCP or UDP. But in the case of network functions, the application is networking. So, we need access to all of those packet headers, which is why AF_XDP is great for us because you get raw packet access. To do that with the Linux kernel, you'd have to use an AF_PACKET socket or something like that, which is-- performance for small packets is not really there, there's a lot of overhead in the kernel stack that you're not using, extra buffer allocation, copies et cetera. I know there's ways to work around that. But for the most part, you're just not getting that. You just don't have access to what you need in the kernel to write these networking applications in user's pace.

The second part of that is the kernel has been around for a very long time, and it's challenging to implement things at the rapid pace that is required to create and deploy these new network services. So, to get all that functionality implemented and integrated into the kernel, and then released on a cadence that is consumable, that lead time tends to be pretty long. So, doing these kinds of applications in userspace allows you to rapidly iterate, try new things, and deploy much quicker.

Xiaojun Li

Great. Thank you. Second question. "Do I need specialized hardware to run the CNDP applications?"

Jeff Shaw

No specialized hardware is required. Like I said, AF_XDP is generic, and it runs as long as your Linux kernel supports AF_XDP. You can run anywhere. So, no special hardware is required.

Xiaojun Li

Great. The next one. "Does CNDP provide hardware drivers for network cards or other accelerators?"

Jeff Shaw

So, that's a good question. So, what we're avoiding in CNDP is having a hardware device driver layer, because that adds a whole lot of extra stuff like PCI bus scanning, you have to manage these devices. So, really, what we want to be able to do is reuse all of the control functionality that's already present in all of these existing Linux drivers, or whatever OS you plan on using. We want to be able to reuse everything that's already there, and really just carve out essentially just queues for data plane, which is what AF_XDP is really good for. So, we want to avoid having specialized hardware drivers. We want to encourage-- if you have a hardware device accelerator, encourage the use of easy-to-use APIs that we can just call into or link dynamically against. Doing it that way allows us to really keep CNDP small and simple, and not have a bunch of extra stuff that's required to manage all kinds of hardware. We think that's best probably separated from CNDP.

Xiaojun Li

That's good. That's good. And the next one, "Do I need to use Kubernetes to run CNDP applications?"

Jeff Shaw

So, no, you don't need Kubernetes, so you can just - it's just software libraries, there's nothing inherent that we're using that's provided by Kubernetes, for example, that we need in CNDP. So, you can just run a CNDP application bare metal on your host if you want, if you're just doing packet processing on top of AF_XDP sockets or whatever interface you've chosen. So, no requirement to use Kubernetes, but if you want to deploy in a cloud environment that's probably a good bet.

Xiaojun Li

We received more questions. And the next one, "Is a CNET graph mode..." sorry, just a second. The next one will be "How does AF_XDP relate to RDMA?"

Jeff Shaw

So, as far as I know, so I'm not an AF_XDP expert, but as far as I know, there's really no relation to AF_XDP and RDMA. But I'm not an expert in either, so I could be wrong.

Xiaojun Li

Thank you. The next one, "Does the platform allow the scaling PODs automatically over K8s and development, how the data path load is loaded up between PODs in this case? Any data path load balancer is coming part of the platform".

Jeff Shaw

So, CNDP isn't providing any of the functionality *per se* that does the auto-scaling of applications or something like that. You would have to do that implementation separately in Kubernetes. How's the data path loaded between pods in that case? So, we're not really doing an automatic scaling. Sorry. I'm going back and reading the question that I see.

We're not doing anything special to auto-scale pods automatically. So, that's really a Kubernetes thing. It's not related to CNDP. Data path load balancer coming as part of the platform.

So, the thing with the secondary network is we're not really... it's pretty separate from your existing Kubernetes networking stuff. So, a lot of this is outside of the scope of CNDP. Let's put it that way.

Xiaojun Li

And the next one is a good one. "How does this compare to DPDK in terms of performance?"

Jeff Shaw

So, that's a really good question and it's one we get very frequently. So, Keith, he's our, he's our lead architect, he's also been around DPDK for a long time, too, we like to put it this way. So, if you take a scale of 1 to 10, where Linux kernel small packet performance is 1, and DPDK is a 10, it's the best in class packet processing, small packet processing performance you can get out there, CNDP is probably a 6, 7, or an 8. So, you're not going to get the optimal performance, but you're going to get a much better performance than you would if you're doing like an AF_PACKET socket or something like that. It really is driven by the performance of AF_XDP. So, which we know provides very good performance relative to the existing Linux-based Linux kernel implementation. So, it's not going to be as high performance as DPDK. But it's going to be-- we're trying to strike that balance between flexibility and functionality and performance.

Xiaojun Li

Got it. Thank you. "Is CNET graph node processing based on the VPP?"

Jeff Shaw

So, I think it is. So, I can tell you we ported the graph node infrastructure from DPDK. And if you look at the code, it's obviously very similar to VPP's graph node implementation. So, I'm assuming that that particular implementation has its roots-- at least it's inspired by VPP's graph node processing. But if you look at the implementation in DPDK, and the one that we carried forward into CNDP, it's not exactly the same implementation, but it's very similar. So, you see a lot of the same things that they do in VPP with the four-at-a-time loop, loop unrolling tricks. We do a lot of that. You can see that in the code. So, it's very similar to what VPP's graph nodes are doing, but it's not the VPP's implementation.

Xiaojun Li

And quick question, "Please mention again the CNDP website".

Jeff Shaw

Cndp.io. So, it's cndp.io.

Xiaojun Li

Cndp.io. Thank you. And two more questions so far. "Do you consider using VDPA (vHost data path acceleration) driver?"

Jeff Shaw

No. We do not consider that. So, I can take a note. I'm going to look at that.

Xiaojun Li

Thank you. The last one, "How do you see adoption in hypescalers?"

Jeff Shaw

So, really to get the most benefits from CNDP is we need to have AF_XDP be pervasively supported across all of the different cloud environments. So, we're hoping that AF_XDP continues to be supported. We need to be able to run on... you need to be able to run your cloud applications on all different kinds of environments. So, once that provides you a pod that has a single network interface that's shared between-- that has a single network interface, we have to be able to run CNDP applications on top of that, which is a little complicated because packets... if you have a single queue, and you open an AF_XDP socket on that queue packets are all coming into your CNDP application. So, then you need to inject them back into the kernel, if you want the kernel stack to process that, if you want the existing Kubernetes liveness probes and API stuff to work on that particular container. So, that's a challenge there. So, that's one reason why we use the secondary network. So, we have to make sure that these are all provided by the cloud providers. We're working on that, nothing to report out today.

Xiaojun Li

One more question popped up, “Does CNDP replace the Intel Smart Edge Open?”

Jeff Shaw

No, absolutely not. They're pretty unrelated. In fact, we are-- CNDP is trying to become a part of the Smart Edge Open. So, we're looking at ways that we can accelerate the applications that are deployed using that platform using CNDP. So, that's one area where we might have some intersection in the future, and we're looking into that. But in no way does it replace Smart Edge Open or any other packet processing environment for that matter. It's a separate thing.

Xiaojun Li

And that's all the questions, and appreciate it, Jeff. Appreciate it. Appreciate it for your insightful presentation. You've got something to say, Jeff, sorry.

Jeff Shaw

No, I just wanted to reiterate my gratitude. Thanks to everybody for watching. I think we had a good discussion here. So, I'm looking forward to engaging with everybody in the community and making the CNDP thing something big.

Xiaojun Li

Thank you. And thank you all for joining us today. And please do not forget to give us the rating for the live recording, so that we can continuously improve the quality of our webinars. Please be sure to join us next time, Wednesday, June 1 at 8 a.m. Pacific Time for the seminar, High Speed Packet Processing with the Data Plane Development Kit, DPDK. You can find the link for registration in the Attachments tab.

Thank you again. This concludes our webcast. Thank you.