TECHNOLOGY GUIDE

Intel Corporation

Cloud Native Data Plane (CNDP) - Overview

Authors

Magnus Karlsson

Gary Loughnane

Elza Mathew

Paulina Osikoya

Jeff Shaw

Edwin Verplanke

Keith Wiles

1 Introduction

There is a fast-growing requirement for a data plane software development environment that is better aligned with the cloud native paradigm. Virtualized packet processing applications can be difficult to efficiently automate and orchestrate in a cloud native platform because of their dedicated resource demands, complex software management models (driver, kernel, software releases, and firmware), and difficulty to debug and monitor. These limitations are orthogonal to cloud native design principles. As such, there is not a clear path for migration from a virtual network function (VNF) to a cloud native network function (CNF), leading to the creation of complex deployment and management models to run ported applications and services. These legacy applications were not designed for the cloud native paradigm and are being retroactively fit into this world.

A new cloud native centric data plane framework is needed to meet the needs of high-performance, cloud native packet processing applications and the design principles of this paradigm. This paper introduces such a framework called the Cloud Native Data Plane (CNDP) and proposes an architecture to deploy it in a cloud native platform. CNDP provides a framework to easily deploy packet processing applications across the comms ecosystem to address composability, automatability, scalability, and performance requirements imposed by communications workloads.

This document is part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.

Table of Contents

Figures

Tables

Document Revision History

REVISION	DATE	DESCRIPTION
001	January 2021	Initial release.
002	October 2021	Updated to include extended scope and a detailed description of the device plugin.
003	May 2022	Updated CNDP release to 22.04 and revised the document for public release to Intel® Network Builders.

1.1 Terminology

Table 1. Terminology

ABBREVIATION	DESCRIPTION
ACL	Access Control List
AF_XDP	Address Family eXpress Data Path
API	Application Programming Interface
BPF	Berkeley Packet Filter
Chnl	Channel
CLI	Command Line Interface
CNCF	Cloud Native Computing Foundation
CNDP	Cloud Native Data Plane (CNDP)
CNET	Cloud Networking Stack
CNF	Cloud Native Network Function
CNI	Container Network Interface
CQ	Completion Queue
DP	Device Plugin
DSA	Data Streaming Accelerator
eBPF	extended Berkeley Packet Filter
FD	File Descriptor
FIB	Forwarding Information Base
FQ	Fill Queue
IA	Intel® architecture
Intel® AVX-512	Intel® Advanced Vector Extensions 512
Intel® DLB	Intel® Dynamic Load Balancer
Intel® DSA	Intel® Data Streaming Accelerator
IP/IPv4/IPv6	Internet Protocol Version 4/Version 6
JSON	JavaScript Object Notation
K8s	Kubernetes
HW	Hardware
MBUF	Memory Buffer
NIC	Network Interface Card
PCI	Peripheral Component Interconnect
PF	Physical Function
PMD	Poll Mode Driver
QUIC	Quick UDP Internet Connection
REST	REpresentational State Transfer
RIB	Routing Information Base
RX	Receive
SLA	Service Level Agreement
SW	Software
SR-IOV	Single Root Input/Output Virtualization
TCP	Transport Control Protocol
TX	Transmit
UDP	User Datagram Protocol
UDS	UNIX Domain Socket
UMEM	A region of virtual contiguous memory that is divided into frames of equal size

ABBREVIATION	DESCRIPTION
VNF	Virtual Network Function
XDP	eXpress Data Path
XSK	AF_XDP Socket

1.2 Reference Documentation

Table 2. Reference Documents

REFERENCE	SOURCE
AF_XDP Overview	https://www.kernel.org/doc/html/latest/networking/af_xdp.html#af-xdp
Devlink enhancements for subfunctions management	https://netdevconf.info/0x14/pub/papers/45/0x14-paper45-talk-paper.pdf
Subfunction management using devlink	https://netdevconf.info/0x14/pub/slides/45/sf_mgmt_using_devlink_netdevconf_0x14.pdf
Principles of container-based application design	https://www.redhat.com/en/resources/cloud-native-container-design-whitepaper#:~:text=Keep%20it%20simple%2C%20stupid%20(KISS,Separation%20of%20concerns%20(SoC)
What is Legacy Application Modernization?	https://www.sdxcentral.com/cloud/definitions/what-is-legacy-application-modernization/
Interacting with eBPF Maps	https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html#interacting-with-maps
Introduce preferred busy-polling	https://lwn.net/Articles/837010/

2 Overview

2.1 Cloud Native Data Plane

CNDP is a data plane framework that uses standard Linux interfaces and operating system mechanisms with Intel® technologies to allow applications to be built, deployed, and managed more efficiently by a cloud native platform while also providing uncompromised performance. It provides a user with the following:

- User space libraries for packet processing microservices in the cloud native paradigm. These libraries take advantage of Intel technologies where possible.
- The components needed to deploy a CNDP pod in Kubernetes, which include a device plugin and a container network interface (CNI).
- A user space networking stack that can terminate traffic or allow you to build your application as part of the stack itself.

Figure 1 shows a high-level overview of CNDP and its scope. CNDP uses AF_XDP as its packet I/O layer, bypassing the kernel network stack. It adds libraries for buffer management and other common packet processing functions like routing and flow classification.

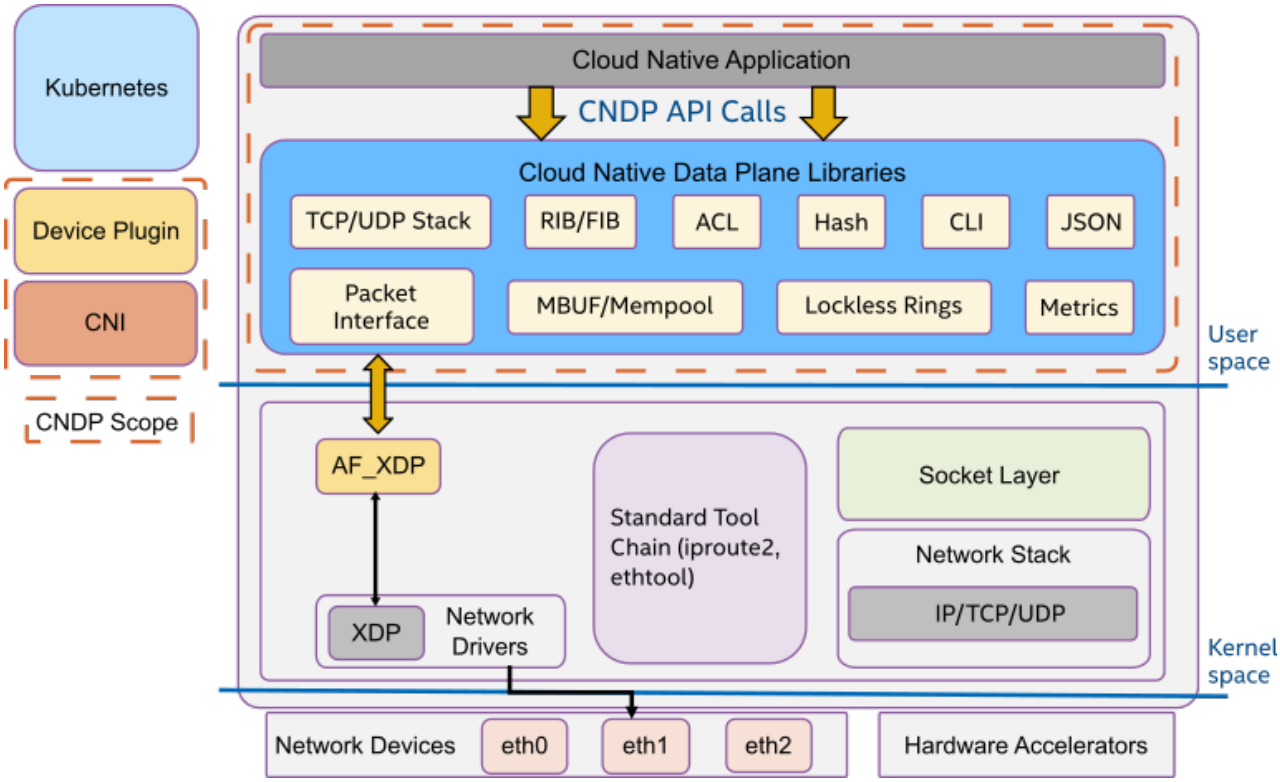

Figure 1: Cloud Native Data Plane Overview

CNDP reuses many of the key learnings and optimizations from the Data Plane Development Kit (DPDK) and adheres to the specification using the cloud native principles, including the following:

- Single Concern Principle (SCP) - every container should address a single concern and do it well.
- High Observability Principle (HOP) – the containerized application must provide APIs for the different kinds of health checks - liveness and readiness.
- Lifecycle Conformance Principle (LCP) - provide APIs for the cloud native platform to read from.
- Image Immutability Principle (IIP) - prevent the creation of similar container images for different environments. One image to rule them all.
- Process Disposability Principle (PDP) - containers need to be as ephemeral as possible and ready to be replaced by another container instance at any point in time.
- Self-Containment Principle (S-CP) - a container should contain everything it needs at build time. It should rely only on the presence of the Linux kernel and have any additional libraries added into it at the time the container is built.
- Runtime Confinement Principle (RCP) – containers should declare required resources (as well as their SLAs) and stick to those SLAs.[1]

CNDP is an open-source, community driven project available at https://cndp.io.

2.2 Challenges Addressed

Similar to the hardware evolution seen with software-defined networking (SDN) and network function virtualization (NFV), software is also evolving. “Legacy application modernization is when an outdated application is updated or rebuilt to effectively work in modern runtime environments and with other applications.”[2] The following table explains how CNDP differs from legacy app modernization:

[1] Paraphrased from “Principles of Container-Based Application Design,” Red Hat, Inc., last updated December 26, 2018, https://www.redhat.com/en/resources/cloud-native-container-design-whitepaper.

[2] Connor Craven, “What Is Legacy Application Modernization?,” SDxCentral, LLC, May 28, 2020, https://www.sdxcentral.com/cloud/definitions/what-is-legacy-application-modernization/.

Table 3. Legacy App (Data Plane) Modernization vs. CNDP[3]

LEGACY APP (DATA PLANE) MODERNIZATION	CLOUD NATIVE DATA PLANE

▪ Monolithic apps
▪ Applications closely coupled with the infrastructure – requires direct access to hardware - no abstraction
▪ Design principles orthogonal to cloud native principles
  ○ Designed to scale up, not scale out
  ○ Specific hardware (HW) requirements – in some cases requires federated zones and specific HW and software (SW) recipes
  ○ Observability, logging, and tracing not built in
  ○ Portability through different environments is difficult
  ○ An update to one part means rolling out a whole new app
▪ Difficult to deploy and manage
	▪ Designed specifically to run in a cloud
	▪ Disaggregates and decouples the application from the infrastructure
	▪ Adheres to cloud native principles
	▪ Loosely coupled, individual and modular microservices
	▪ Have only the elements of an operating system (OS) needed to run without external dependencies
	▪ Observability, tracing, and logging built in
	▪ Configuration through REST APIs
	▪ Portable images through different environments
	▪ Pure SW fallbacks for libraries enable the principle of "running anywhere"
	▪ Designed to scale out
	▪ Integrate with CNCF projects from the beginning to enable seamless integration with Kubernetes

2.3 Technology Description

The goal of CNDP is to provide a framework for packet processing microservices that is better aligned with Kubernetes (public and private cloud) deployments. This includes providing the entities to provision, orchestrate, and manage the data plane (using cutting-edge cloud native practices) and the data plane itself.

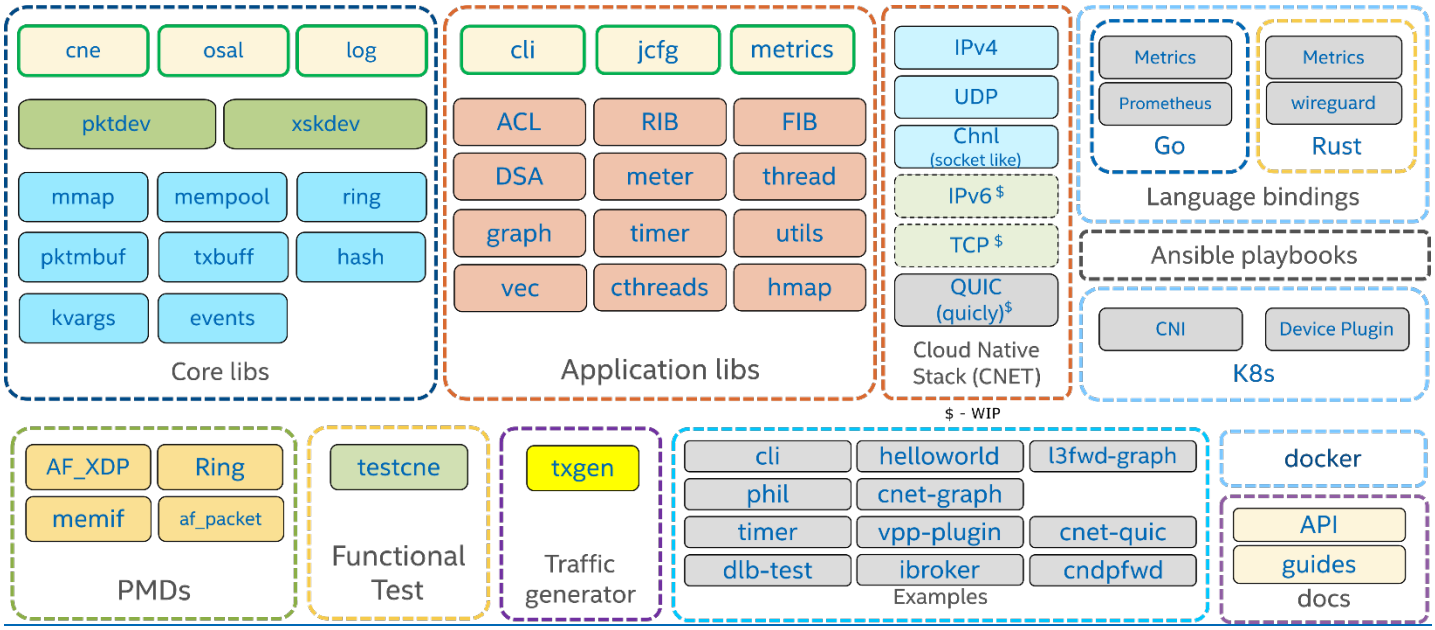

Figure 2: CNDP 22.04 Release

Figure 2 shows a snapshot of the latest CNDP release (v22.04). It provides:

- Core Libraries: Lightweight libraries that provide APIs for managing memory and networking interfaces.
- Application Libraries: Libraries that provide all the support needed to build an application on top of CNDP.
- Poll Mode Drivers: These are abstractions over networking interfaces like AF_XDP. It is important to note that CNDP does not interact directly with any physical network devices.
- Network Stack: User space network stack (CNET) to accelerate transport layer processing.
- A test suite for functional testing.
- Sample applications, including high throughput traffic generator (txgen).
- A Prometheus metrics agent for providing telemetry output.

[3] Workloads and configurations. Results may vary.

- Rust language bindings, including a WireGuard implementation for CNDP.
- Kubernetes and Docker specs to build images and launch CNDP pods.

CNDP debunks the commonly held belief that in order to develop cloud native packet processing microservices you must sacrifice performance (by relying on software to do everything) to gain flexibility and vice versa. With CNDP, we showcase that packet processing microservices can still be performant on Intel® architecture (IA) without sacrificing cloud native principles. This is accomplished with a combination of IA technologies and Kubernetes enablement.

- IA: Enable Intel technologies, such as Intel® Data Streaming Accelerator (Intel® DSA), Intel® Dynamic Load Balancer (Intel® DLB), and Intel® Advanced Vector Extensions 512 (Intel® AVX-512), in CNDP to accelerate the various libraries and algorithms used by packet processing applications. Note: There will always be a software fallback for enabled features.
- Kubernetes: Create CNDP operators, device plugins, and CNIs in Kubernetes that can orchestrate (provision, advertise, and manage) all the resources that can be used by CNDP applications as well as manage the application itself.
- Network Stack: Provides sockets-like interface with multi-packet batching and zero-copy interface. Interface addresses, routes, and neighbors managed by Linux and learned via Netlink. The latest CNDP release supports IPv4/UDP, with IPv6 and TCP to be added in future releases.

One of the networking interfaces supported by CNDP is AF_XDP. AF_XDP allows us to meet the performance needs for a cloud native data plane.

2.4 AF_XDP Deep Dive

AF_XDP socket (XSK) is a new type of socket that is optimized for high performance packet processing. It takes advantage of an in-kernel fast path called eXpress Data Path (XDP). XDP is an eBPF program that can redirect packets directly from the NIC to the user space application through an AF_XDP socket using the XDP_REDIRECT action. An AF_XDP socket is created with the normal socket() syscall. Associated with each XSK are two rings: the receive (RX) ring and the transmit (TX) ring. A socket receives packets on the RX ring and sends packets on the TX ring. An RX or TX descriptor ring points to a data buffer in a memory area called a UMEM.[4]

For more information about AF_XDP, see AF_XDP Sockets: High Performance Networking for Cloud-Native Networking Technology Guide and AF_XDP – In Kernel Fast Path Overview Training Video.

2.4.1 UMEM

The UMEM consists of several equally sized chunks (frames). A descriptor in one of the rings references a frame by its offset within the entire UMEM region. The user space application allocates memory for this UMEM by whatever means it feels is most appropriate, for example: malloc, mmap, or HugePages. It is mapped between kernel and user space to provide a zero-copy interface.

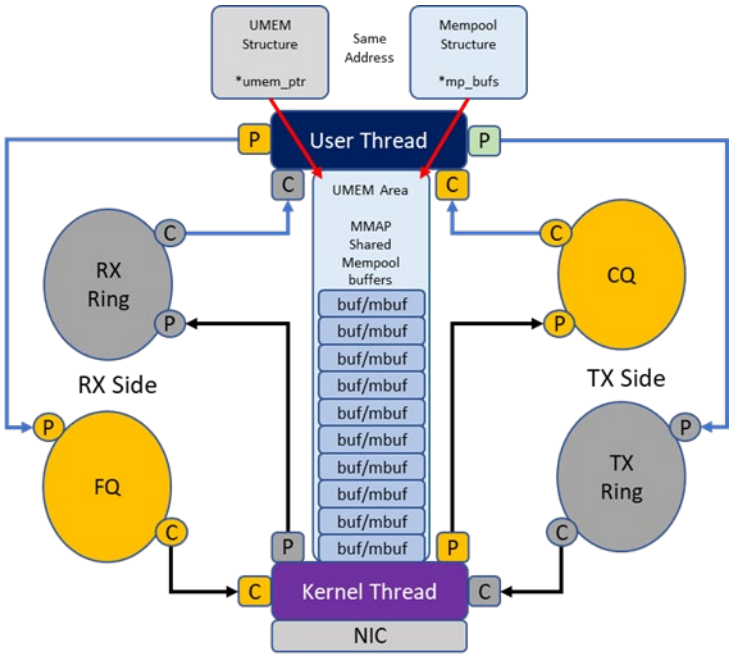

Figure 3: AF_XDP Overview

[4] Workloads and configurations. Results may vary.

The UMEM also has two rings: the FILL queue (FQ) and the COMPLETION queue (CQ). The FILL queue is used by the application to send addresses for the kernel to fill in with RX packet data. References to these frames appear in the RX ring after each packet has been received. The COMPLETION queue contains frame addresses that the kernel has transmitted completely and can be used again by user space, for either TX or RX. Thus, the frame addresses appearing in the COMPLETION queue are addresses that were previously transmitted using the TX ring. In summary, the RX and FILL rings are used for the RX path and the TX and COMPLETION rings are used for the TX path.

The socket is then finally bound with a bind() call to a device and a specific queue id on that device, and it is not until bind is completed that traffic starts to flow. The XSK socket creation and loading of the XDP eBPF program is exposed to a user space application through libbpf API.

2.4.2 Busy Poll

"AF_XDP can be interrupt driven or busy-poll based. With busy-poll, the driver is executed in process context by calling the poll() syscall. The main advantage with this is that all processing occurs on a single core. This eliminates the core-to-core cache transfers that occur between the application and the softirqd processing on another core, that occurs without busy-poll. From a systems point of view, it also provides an advantage that we do not have to provision extra cores in the system to handle ksoftirqd/softirq processing, as all processing is done on the single core that executes the application. The drawback of busy-poll is that max throughput seen from a single application will be lower (due to the syscall), but on a per core basis it will often be higher as the normal mode runs on two cores and busy-poll on a single one."[5]

Socket options need to be configured by a privileged entity. The CNDP K8s device plugin handles that programming. The AF_XDP socket creation is handled by the CNDP application except for the configuration of busy polling on the socket and the loading of the BPF program. Since the configuration of the busy polling and the loading of the BPF program are privileged operations, the CNDP application relies on the K8s device plugin to perform these operations on behalf of the CNDP application.

The socket file descriptor is passed from the CNDP application to the K8s device plugin along with values for the busy_timeout and the busy_budget that are used by the setsockopt() calls. A message is sent from the CNDP application to the K8s device plugin, /config_busy_poll, $socket_fd, $busy_timeout, $busy_budget.

The K8s device plugin responds with a /config_busy_poll_ack message if the configuration of busy polling on the socket was successful. On failure, the K8s device plugin responds with a /config_busy_poll_nak message.

2.4.3 AF_XDP Abstraction

CNDP provides two APIs to abstract the lower-level details of the XSK APIs: xskdev and pktdev (Figure 4). The xskdev API provides a set of wrappers around the XSK APIs. The pktdev API is the highest level of abstraction. It provides an API to manage multiple port types including AF_XDP and ring-based ports. Memory pool and buffer management are built into this API.

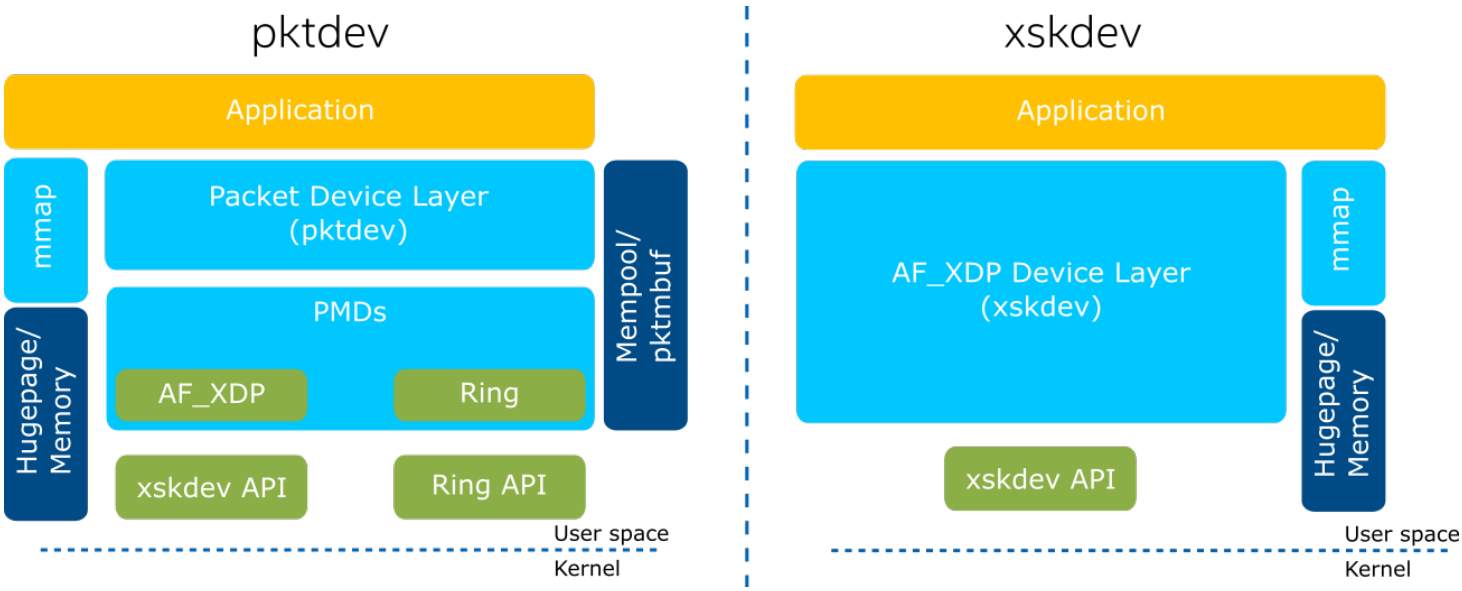

Figure 4: CNDP Interface APIs

[5] Karlsson, Magnus. "busy poll support for AF_XDP sockets". Netdev Mailing List. https://patchwork.ozlabs.org/project/netdev/cover/1556786363-28743-1-git-send-email-magnus.karlsson@intel.com/

With CNDP, the application allocates the memory, from HugePages or somewhere else, and uses this area to create a mempool. The mempool divides the memory area into equal size packet buffers (pktmbufs) that are eventually passed to the XSK APIs, and mmap is used to register the memory (referred to as UMEM) with the kernel. This is shown in Figure 3.

3 Deployment

Figure 5 shows the current deployment model for an application pod that uses AF_XDP. Note that the Sidecar Container shown is optional.

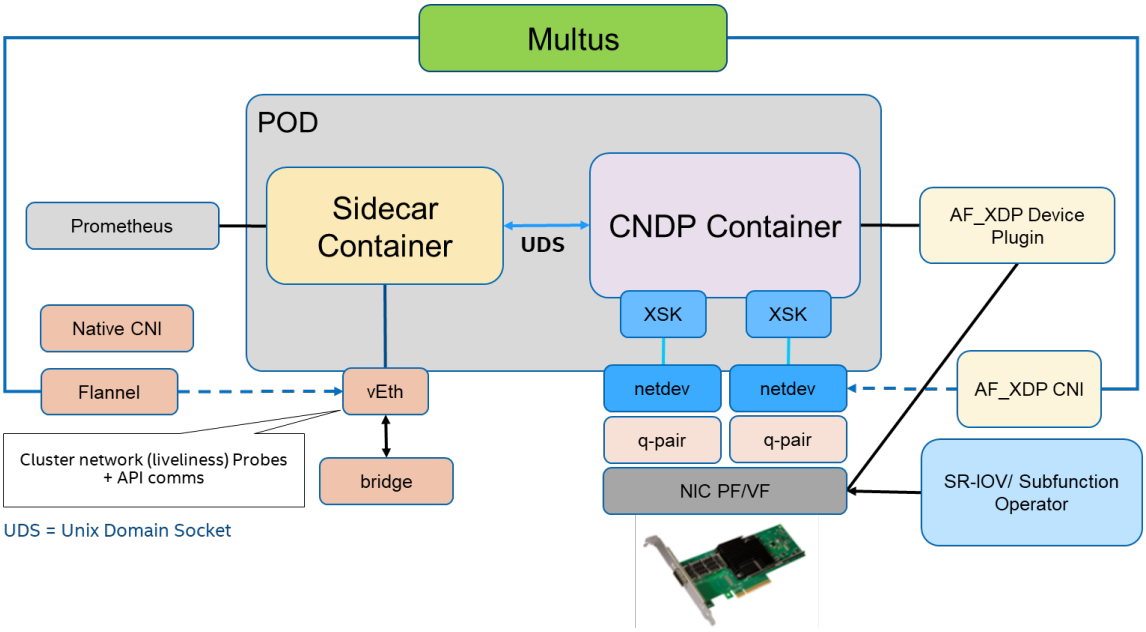

Figure 5. Current AF_XDP Deployment Model

A CNI plugin is a binary invoked by the Kubernetes kubelet during pod creation. CNI plugins are responsible for the network provisioning of the pod. This is accomplished by moving a netdev from the host network namespace into the pod network namespace. It should be noted that all containers within a pod share a network namespace, meaning the attached netdev is available to all containers within the pod. The host device CNI (in conjunction with Multus) attaches an additional (whole) netdev to the AF_XDP pod. This is the netdev that is used to create AF_XDP sockets. The user/application is expected to program netfilter rules on the netdev before/after it is attached to the pod networking namespace. Flannel is used to attach a vEth interface to the pod for cluster network probes.

3.1 AF_XDP Deployment Model Issues

Today, XDP is designed for use by infrastructure rather than by applications in the Kubernetes domain. Thus, several limitations exist with the model to provision and launch and manage an AF_XDP pod.

- The AF_XDP container must run as a privileged container to create the AF_XDP socket.
- With the existing CNI (HostDevice), dynamic allocation of netdevs is limited to devices with a PCI address. The netdev remains in a DOWN state after attachment to the pod and the netdev is also renamed because the HostDevice CNI honors the Multus-provided IfName. This is an issue as the interface name is considered as the unique identifier of the netdev since ifindex can be inconsistent across namespaces.
- The Ethtool/netfilter rules that are configured on a netdev and are used to direct packets to different hardware queues are persistent regardless of the networking namespace they are attached to. These should be cleared as the netdev is moved from one network namespace to another.
- Ethtool/netfilter configuration requires privilege.
- A full networking interface, rather than a queue pair from a port, must be passed to the CNDP pod resulting in inefficient resource utilization.
- The ifindex, which is the identifier given by the kernel to identify a network device, is not unique across Linux network namespaces and can change if there is an ifindex clash. eBPF program loading and unloading relies on ifindex – as such there is a need for tracking the ifindex across networking namespaces.

3.2 CNDP Deployment Model and Proposed Solutions to Deployment Issues

Figure 5 shows the CNDP solution for deploying a cloud native application based on AF_XDP. The CNDP deployment model includes a new device plugin and CNI that address the issues with the AF_XDP deployment model. These go hand in hand with modification to the kernel to overcome the issues when creating AF_XDP sockets. The subfunction API is used to slice the NIC into smaller resource sets (netdevs) that can be attached to pods separately. A device plugin is used to load the eBPF program in the host namespace before the pod is launched. The CNI is used to program network filters on the PF. After the CNDP pod is launched, the XSK_MAP FD is passed from the device plugin to the CNDP pod (over a shared UNIX Domain Socket) so that it can create the AF_XDP sockets that it needs without the need for any extra privileges. This model requires that the netdev name (ifname) is fixed across different networking namespaces so that there is a map of consistent interfaces regardless of ifindex clashes.

The CNDP pod consists of two containers:

- An application container that runs the CNDP application
- A sidecar container that runs auxiliary functionalities like interfacing with Prometheus or the REST API interface that allows for configuring/sending control messages to the CNDP application

The two containers communicate over a UNIX Domain Socket (UDS). Using a sidecar pattern allows for the independent scaling of CNDP data plane containers from the tools exporting telemetry, logging, and presenting REST APIs from/for those containers.

3.2.1 Privileged AF_XDP Container

Privilege is the biggest challenge for any application that wants to use AF_XDP in a pod without privilege. Running pods in privileged mode gives containers the same access as processing running on the host, which is generally not needed for containerized applications and is less secure than running containers without privilege. The solution here involves three parts:

- Firstly, break up the AF_XDP socket creation into the loading of the eBPF program and then the creation of the AF_XDP socket. Doing this enables the privileged functionality (loading the program) to be handled by an entity that has root privileges outside of the pod itself and can do so as part of the pod deployment process.
- Secondly, during the process of creating an AF_XDP socket, retrieve the XSK_MAP file descriptor (FD) so that it can be populated with the XSK (to inform the eBPF program where to redirect packets). If the eBPF program is loaded outside of the pod, this file descriptor must be passed to the CNDP container by another live process.
- Thirdly, if the socket is a busy polling socket, then also rely on the device plugin to configure the busy-poll socket option as this is a privileged operation. Device Plugin Sequence Flows shows how this interaction takes place.

It is also important to note that typically the eBPF MAP file descriptor can be passed from one process to another in one of two ways:

1. Through a UNIX Domain Socket (UDS). The process that passes the file descriptor needs to be active (alive) until the transfer is complete. This means that an Init container cannot be used as it will terminate before all other containers in the pod are started.
2. Export the map to a special eBPF file system (persistent eBPF maps. Note that this file system is located at `/sys/fs/bpf` and mounting this location into an unprivileged pod requires that you relax the default pod security policy for unprivileged pods.

For CNDP, a device plugin is used to load the eBPF program and pass the XSK_MAP FD to the CNDP container via UDS. Modifications to break up the eBPF program loading from the XSK socket creation were also submitted to the kernel to enable this.

Note: UDS communications can be secured by using projects such as Pod2Daemon. The UDS can be used to ensure pod identity and, in the future, to request additional netfilter configurations for the parent netdev of a subfunction.

3.2.2 AF_XDP Device Plugin and CNDP CNI

The device plugin handles the task of loading the eBPF program and providing the XSK MAP FD to the CNDP container after the pod has started. The device plugin is stateful and stays running after the pod has started. The AF_XDP CNI compliments the AF_XDP device plugin. It has the comparatively simple task of moving the netdev into the pod namespace, but, crucially, it can do this dynamically based on the netdev name provided by the device plugin. It does not rename the netdev and the netdev remains in the UP state. The CNI also has a role with setting the appropriate ethtool filters. It is also important to understand CNIs are stateless, invoked only during pod creation and deletion. If the CNI returns an error, then kubelet will not start the pod.

3.2.2.1 Ethtool/Netfilter Solution

The programming and clearing of the netfilters to redirect the desired traffic to the correct AF_XDP port is accomplished through the CNI based on the destination IP address. If the application wishes to add additional filters, it needs to request the device plugin to program the netfilters on the parent netdev.

Note: Until CNDP includes support for the subfunction API, you can use an init container to program the desired extended netfilters on the netdev as it is in the pod namespace. The optimal solution involves the use of an operator that programs extended filters for a pod on the parent netdev.

3.2.2.2 Inability to Slice a PF into Smaller netdevs

Proposed enhancements to the devlink API in the Linux kernel for subfunctions management will support the ability to create, configure, and deploy a much more granular portion of a device from a NIC[6]. That is, "a new light weight PCI function and its associated class devices"[7] – "aka as 'slice'."[8] This API will enable the efficient slicing of a PF into netdev-queue pairs that can be efficiently allocated to a pod. Until those enhancements are enabled, a virtual function can be used to slice up a netdev to share among multiple pods.

3.2.2.3 ifindex Clashes Across Namespaces

The ifindex is used to load/unload the eBPF program. As a netdev is moved in and out of different networking namespaces, the ifindex that identifies it can change (if there is a clash with the ifindex of another netdev in that namespace). The proposed solution uses the device plugin to provision and maintain unique interface names that are used in both the host and the pod. That way the interface can be tracked even if the ifindex changes across namespaces. The correct ifindex can be retrieved in any namespace by calling: `if_nametoindex(ifname);`

3.2.2.4 Unloading eBPF Program

When a pod is deleted, the CNI returns the netdev to the host network namespace and clears any netfilters. During pod creation the device plugin is tasked with loading the eBPF program onto the netdev before the CNI moves it into the pod namespace. During pod deletion, however, unloading of the eBPF program is the responsibility of the CNI. The CNI is chosen because it has the advantage of being hooked into the Kubernetes lifecycle at this point. Also, the CNI has a delete function while the device plugin can only be called upon during pod creation.

3.2.3 Device Plugin Sequence Flows

The following sections provide a deep dive into the sequence flows from device plugin initialization all the way through to pod deletion.

Device Plugin Initialization:

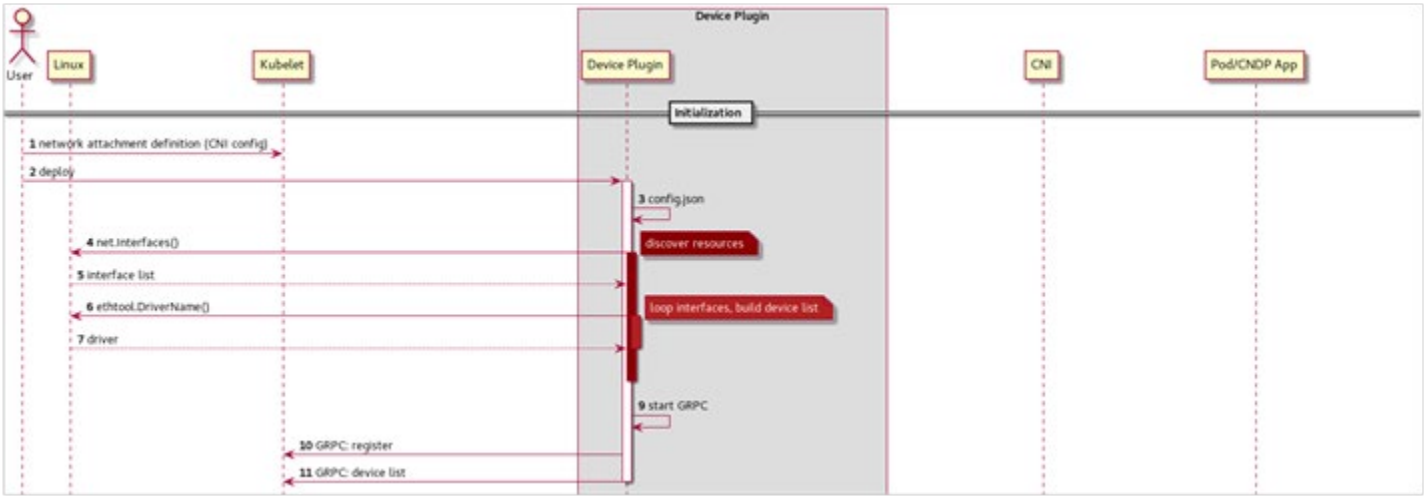

Figure 6: Device Plugin Initialization

[6] Pandit, Parav, "Devlink enhancements for subfunctions management," NetDev Society, accessed December 2020, https://netdevconf.info/0x14/pub/papers/45/0x14-paper45-talk-paper.pdf.

[7] Pandit, Parav, "subfunction management using devlink," NetDev Society, August 18, 2020, https://netdevconf.info/0x14/pub/slides/45/sf_mgmt_using_devlink_netdevconf_0x14.pdf.

[8] Pandit, Parav, "subfunction management using devlink."

Pod Creation:

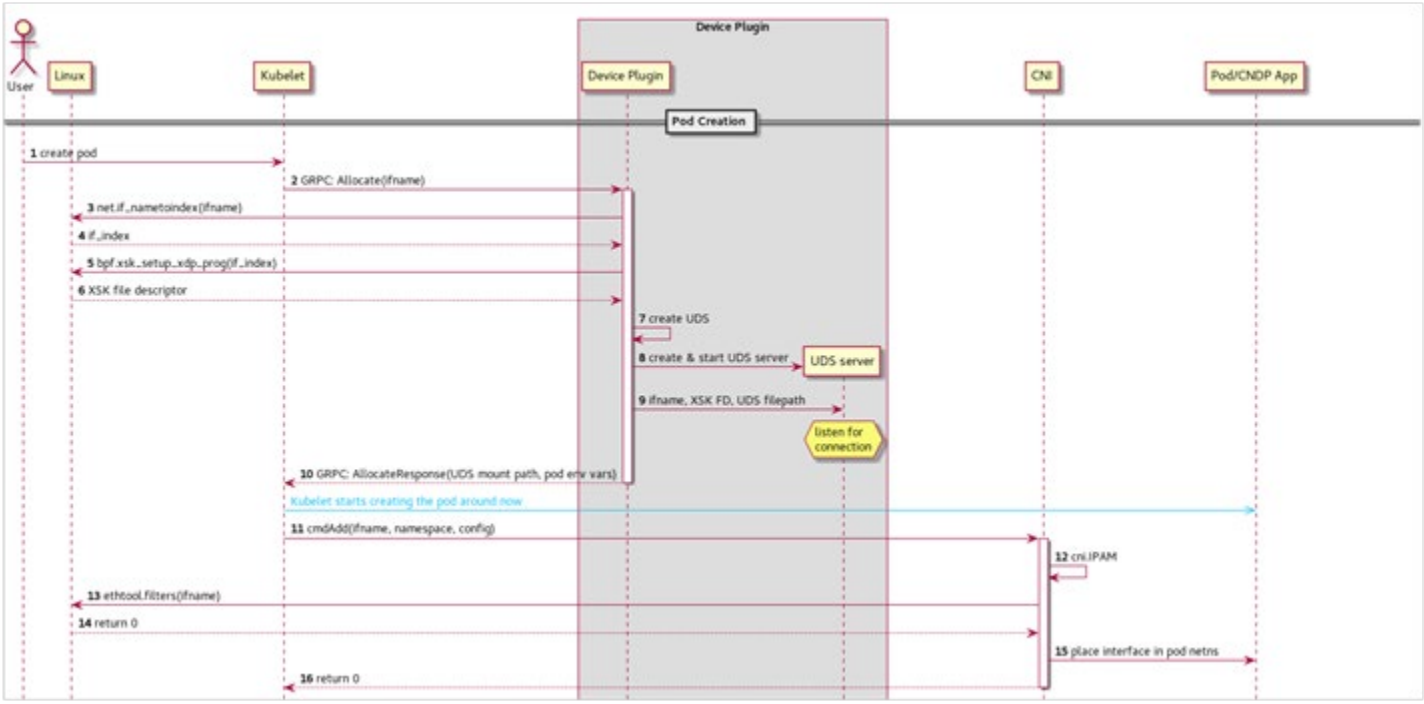

Figure 7: AF_XDP Device Plugin Interactions at Pod Creation Time

Pod Running:

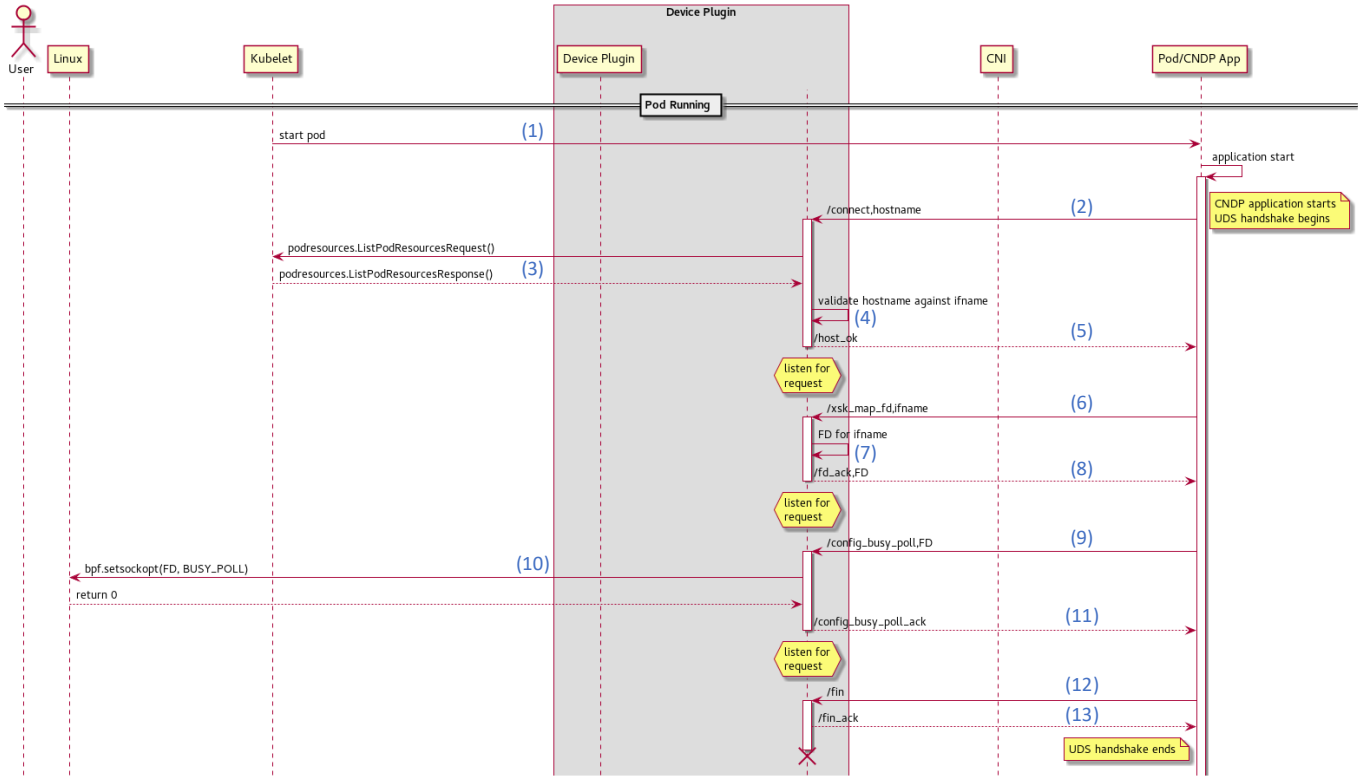

Figure 8: AF_XDP Socket Creation with CNDP

1. The pod is started and the application launches.

- On application launch, a script runs that gathers local information (netdevs, cpus, ...) and generates a jsonc configuration file that is consumed by the CNDP application.

2. CNDP connects to the UDS (which is always in the same path from the container point of view) and initiates a handshake. Note: The Hostpath[9] configuration used to mount the UDS into the container is generated dynamically as part of the pod deployment (see Figure 8).
3. The device plugin (DP) checks what resources (netdevs) are allocated to the pod. At this point, the DP is already aware of what netdevs it is serving. It has the xsk_map FDs ready and waiting (which were retrieved at pod creation time). However, it is not (yet) aware of what pod it is serving. The DP contacts the K8s pod resource's API, which returns a map of all pods and attached devices for the node.
4. The device plugin validates that the correct pod is connected to the UDS and the device plugin is requesting the resource for an appropriate netdev using the map retrieved in step 3.
5. The device plugin acks and waits for the next request from CNDP.
6. When CNDP tries to create and AF_XDP socket, it realizes that it needs to retrieve the xsk_map_fd. The app requests the xsk_map_fd for a specific interface from the device plugin.
7. The device plugin retrieves the xsk_map_fd for that interface.
8. The device plugin sends the file descriptor to the CNDP application.
9. If the CNDP application needs to configure busy polling for a socket, it sends that socket_fd to the device plugin.
10. The device plugin sets the appropriate sockopt to enable busy polling.
11. The device plugin acks the busy polling configuration.
12. CNDP sends a fin when it completes interacting with the device plugin.
13. The device plugin acks.

Pod Deletion:

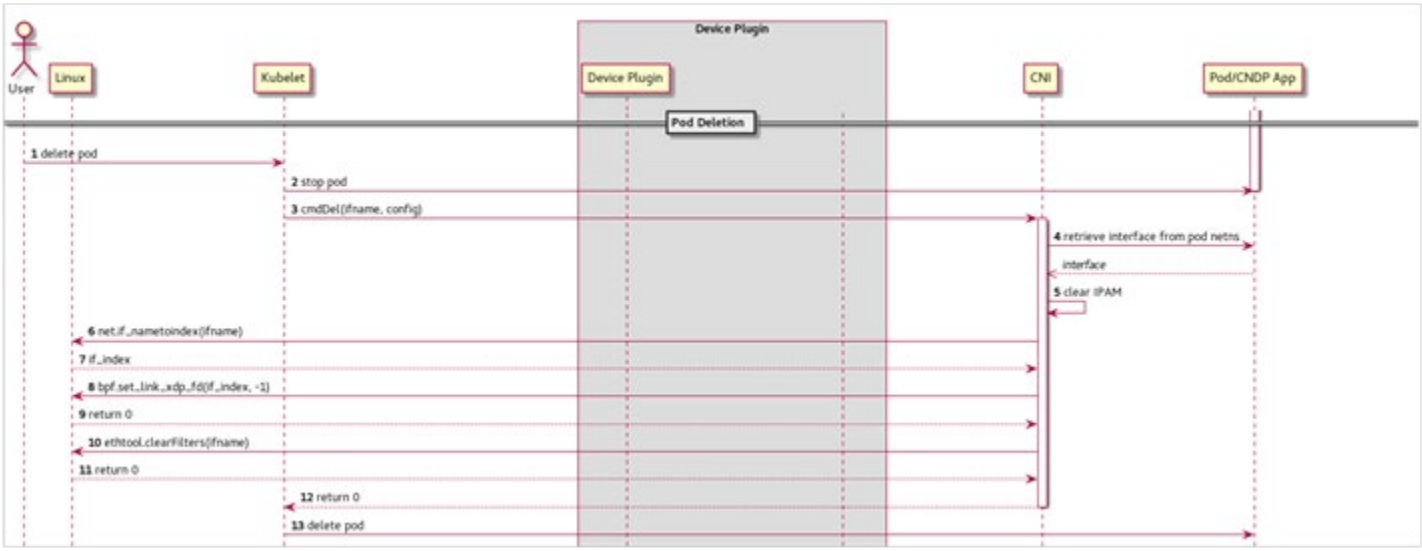

Figure 9: Pod Deletion Flow

4 Summary

Packet processing applications can be difficult for a cloud native platform to efficiently automate and orchestrate. CNDP is a purpose-built cloud native data plane to help address and overcome these difficulties. It provides a framework for packet processing microservices that is better aligned with Kubernetes (public and private cloud) deployments. This includes providing the entities to provision, orchestrate, and manage the data plane using cutting-edge cloud native practices as well as the data plane itself. CNDP is built on top of standard Linux libraries and takes advantage of Intel technologies to accelerate where possible. CNDP is an open-source, community driven project available at https://cndp.io.

You can download CNDP from https://github.com/CloudNativeDataPlane/cndp.

[9] Kubernetes, Volumes. https://kubernetes.io/docs/concepts/storage/volumes/#hostpath

0522/DN/WIPRO/PDF 634083-003US