Software Networking

The Intel Infrastructure Processing Strategy

Francesco Barletta, Luca Rocco

January 13, 2023

Contents

1 Intel® Infrastructure Processing Unit

1.1 What is an IPU?

In a typical "server optimized" enterprise data center, systems are designed for use by a single party, that is, the enterprise itself. However, in a CSP (*Communications Service Providers*) cloud data center, the workload is owned by the tenant, while the systems themselves are owned by the Service Provider.

Further, in highly virtualized environments, significant amounts of server resource are expended processing tasks beyond user applications, such as hypervisors, container engines, network and storage functions, security, and vast amounts of network traffic.

To address this challenge Intel has introduced a new class of product called the IPU. An IPU is an advanced networking device with hardened accelerators and Ethernet connectivity that accelerates and manages infrastructure functions using tightly coupled, dedicated, programmable cores. An IPU offers full infrastructure offload and provides an extra layer of security by serving as a control point of the host for running infrastructure applications. By using an IPU, the overhead associated with running infrastructure tasks can be offloaded from the server (Figure 1).

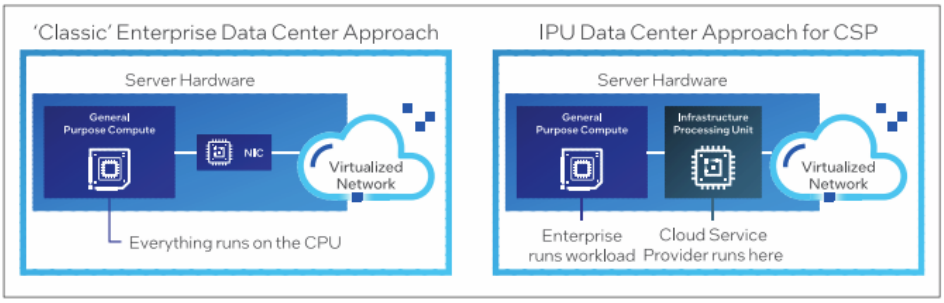

Figure 1: IPU 'disaggregation' in the CSP data center

In other words, the CSP software runs on the IPU itself, while the tenant's applications run on the server CPU. This not only frees up resources on the server, whilst optimizing overall performance, but provides the CSP with a separate and secure control point.

It's also important to note the difference between an IPU and a SmartNIC. A SmartNIC is a programmable network adapter that can accelerate infrastructure applications, however, unlike an IPU it does not provide offload capability to run the entire infrastructure stack and therefore does not give the service provider an extra layer of security and control, enforced in hardware.

1.2 How does an IPU work?

As data center networking marches forward from 25 GbE, to 50 GbE and into the realm of Terabit Ethernet (100+ GbE), it creates unprecedented volumes of network traffic. The net result is an exponential increase in the number of packets transferred per second putting incremental strain on the capabilities of a traditional Network Interface Card (NIC).

Additionally, the advent of software-defined networking (SDN) puts more load onto servers as CPU cores are swallowed up with virtual switches, load balancing, encryption, deep packet inspection, and other I/O intensive tasks. Add into the mix the increasing sophistication of management software running on servers, and it becomes evident that there is a genuine need to manage the explosive growth in network traffic while also offloading "infrastructure" workloads from server CPUs to enable more resources to be dedicated to mission-critical application processing. To put this into context, studies have shown that networking in highly virtualized environments can consume upwards of 30 percent of the host's CPU cycles [1].

IPUs combine hardware-based data paths, which can include FPGAs, with processor cores. This enables infrastructure processing at the speed of hardware to keep up with increasing network speeds and the flexibility of software to implement control plane functions. With the development of its first IPU, Intel has combined onto a single card an Intel Stratix 10 FPGA, through which a highspeed Ethernet controller and programmable data path is implemented, along with an Intel Xeon D processor for the control plane functions.

Blending this capability with the ongoing trend in microservices development offers a unique opportunity for function-based infrastructure—achieved through matching optimal hardware components and common software frameworks to each application or service. For the CSP, this represents an opportunity to accelerate the cloud while hosting more services (apps/virtual machines) on a single machine, leading to improved service delivery and greater profit potential per server.

Also, An IPU has dedicated functionality to accelerate modern appli-

cations that are built using a microservice-based architecture in the data center. As a result, a cloud provider can securely manage infrastructure functions while enabling its customer to entirely control the functions of the CPU and system memory.

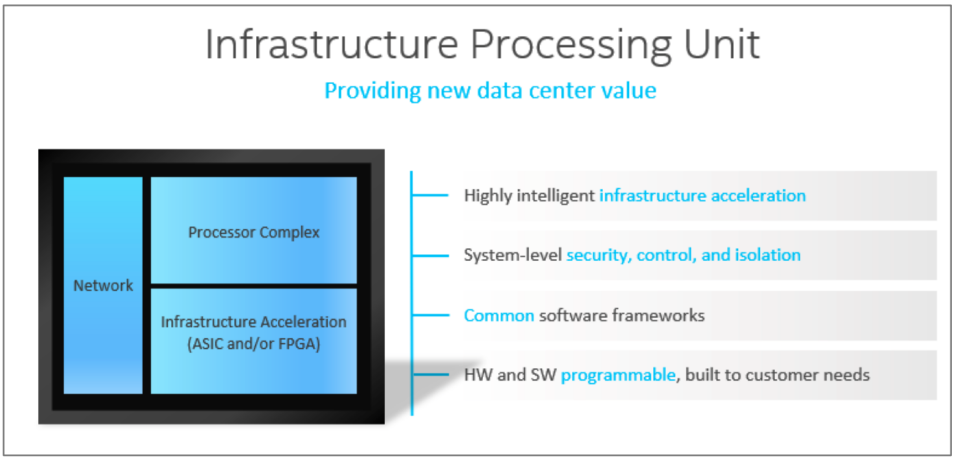

Figure 2: IPU conceptual architecture

1.3 What are the main features of an IPU?

There are four main features of an IPU (Figure 2):

1. Highly intelligent infrastructure acceleration
2. System-level security, control, and isolation
3. Common software frameworks
4. Programmable hardware and software, built to the customer's needs

With these features, an IPU has the ability to:

1. Accelerate infrastructure functions, including storage virtualization, network virtualization, and security with dedicated protocol accelerators.
2. Free up the CPU by shifting storage and network virtualization functions that were previously done in software on the CPU to the IPU.

3. Improve data center utilization by allowing for flexible workload placement.

4. Enable cloud service providers to customize infrastructure function deployments at the speed of software.

On a larger scale, evolving data centers will require a new intelligence architecture where large-scale distributed compute systems work together seamlessly connected as a single platform (Figure 3). This will help resolve today's challenges of stranded resources, congested data flow, and incompatible platform security. Within this new architecture, there will be three categories of compute:

- the CPU for general-purpose computing
- the XPU (cross-platform unit) for application-specific or workload-specific acceleration
- the IPU for infrastructure acceleration

All three categories will be connected through programmable networks to efficiently utilize data center resources.

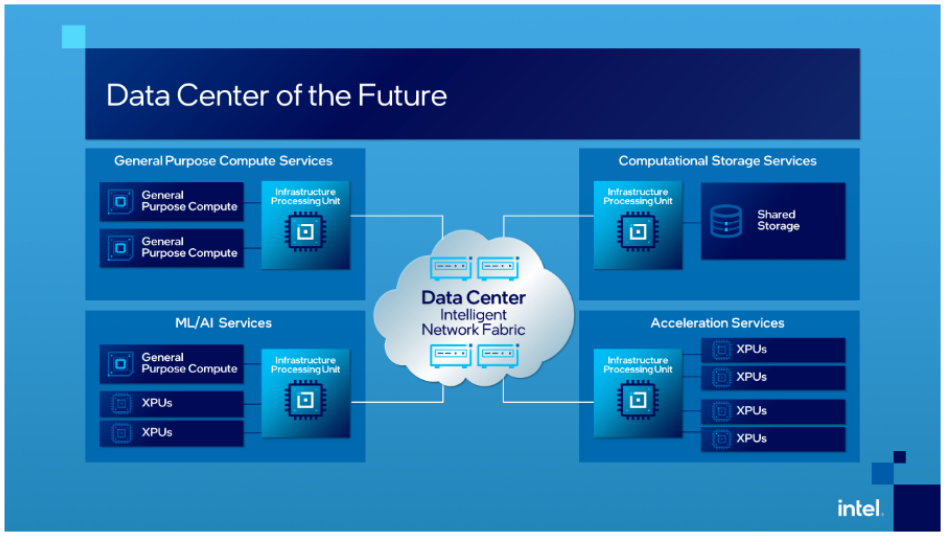

Figure 3: Evolving data centers connected as a single platform.

2 P4-Open vSwitch

2.1 What is Open vSwitch?

The device used in the Intel solution is the P4 OvS. Open vSwitch is a multilayer software switch licensed under the open source Apache 2 license. The goal is to implement a production quality switch platform that supports standard management interfaces and opens the forwarding functions to programmatic extension and control (Figure 4).

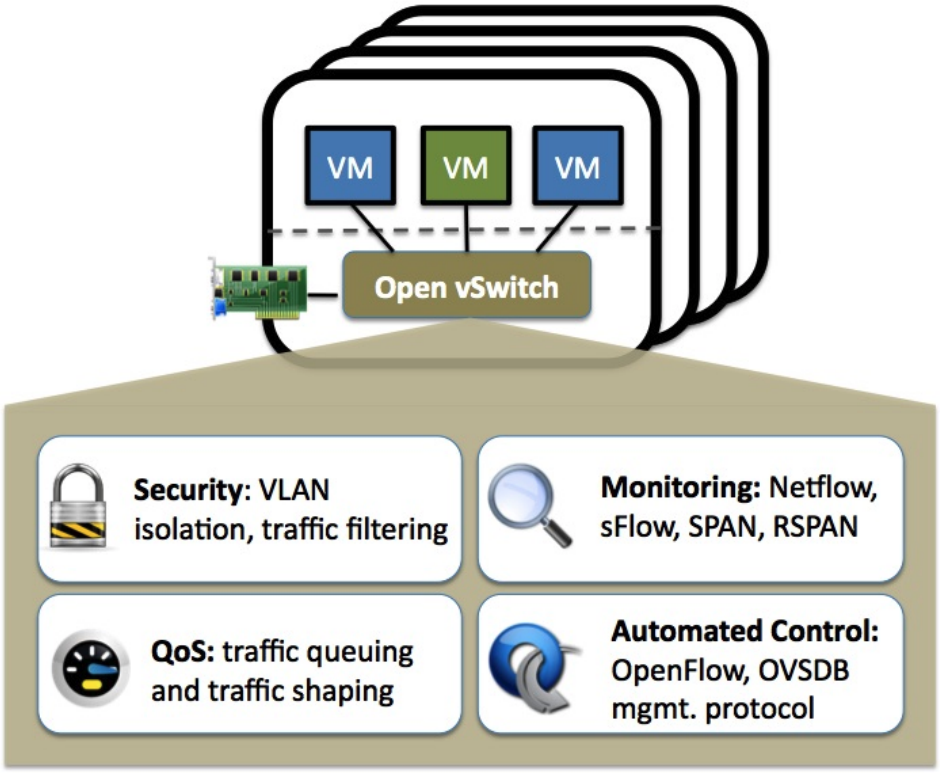

Figure 4: Overview of OvS

Open vSwitch is well suited to function as a virtual switch in VM environments. In addition to exposing standard control and visibility interfaces to the virtual networking layer, it was designed to support distribution across

multiple physical servers. Open vSwitch supports multiple Linux-based virtualization technologies including KVM, and VirtualBox [2].

The bulk of the code is written in platform-independent C and is easily ported to other environments. The current release of Open vSwitch supports the following features:

- Standard 802.1Q VLAN model with trunk and access ports
- NIC bonding with or without LACP on upstream switch
- NetFlow, sFlow(R), and mirroring for increased visibility
- QoS (Quality of Service) configuration, plus policing
- Geneve, GRE, VXLAN, STT, and LISP tunneling
- 802.1ag connectivity fault management
- OpenFlow 1.0 plus numerous extensions
- Transactional configuration database with C and Python bindings
- High-performance forwarding using a Linux kernel module

The included Linux kernel module supports Linux 3.10 and up. Open vSwitch can also operate entirely in userspace without assistance from a kernel module. This userspace implementation should be easier to port than the kernel-based switch. OVS in userspace can access Linux or DPDK devices. Note Open vSwitch with userspace datapath and non DPDK devices is considered experimental and comes with a cost in performance.

2.2 What's inside?

The main components of this distribution are:

- ovs-vswitchd, a daemon that implements the switch, along with a companion Linux kernel module for flow-based switching.
- ovsdb-server, a lightweight database server that ovs-vswitchd queries to obtain its configuration.
- ovs-dpctl, a tool for configuring the switch kernel module.
- Scripts and specs for building RPMs for Red Hat Enterprise Linux and deb packages for Ubuntu/Debian.

- ovs-vsctl, a utility for querying and updating the configuration of ovs-vswitchd.
- ovs-appctl, a utility that sends commands to running Open vSwitch daemons.

Open vSwitch also provides some tools:

- ovs-ofctl, a utility for querying and controlling OpenFlow switches and controllers.
- ovs-pki, a utility for creating and managing the public-key infrastructure for OpenFlow switches.
- ovs-testcontroller, a simple OpenFlow controller that may be useful for testing (though not for production).
- A patch to tcpdump that enables it to parse OpenFlow messages.

2.3 Why Open vSwitch?

Hypervisors need the ability to bridge traffic between VMs and with the outside world. On Linux-based hypervisors, this used to mean using the built-in L2 switch (the Linux bridge), which is fast and reliable. So, it is reasonable to ask why Open vSwitch is used.

The answer is that Open vSwitch is targeted at multi-server virtualization deployments, a landscape for which the previous stack is not well suited. These environments are often characterized by highly dynamic end-points, the maintenance of logical abstractions, and (sometimes) integration with or offloading to special purpose switching hardware. The following characteristics and design considerations help Open vSwitch cope with the above requirements.

2.3.1 Mobility of state

All network state associated with a network entity (say a virtual machine) should be easily identifiable and migratable between different hosts. This may include traditional "soft state" (such as an entry in an L2 learning table), L3 forwarding state, policy routing state, ACLs, QoS policy, monitoring configuration (e.g. NetFlow, IPFIX, sFlow), etc.

Open vSwitch has support for both configuring and migrating both slow (configuration) and fast network state between instances. For example, if

a VM migrates between end-hosts, it is possible to not only migrate associated configuration (SPAN rules, ACLs, QoS) but any live network state (including, for example, existing state which may be difficult to reconstruct). Further, Open vSwitch state is typed and backed by a real data-model allowing for the development of structured automation systems.

2.3.2 Responding to network dynamics

Virtual environments are often characterized by high-rates of change. VMs coming and going, VMs moving backwards and forwards in time, changes to the logical network environments, and so forth.

Open vSwitch supports a number of features that allow a network control system to respond and adapt as the environment changes. This includes simple accounting and visibility support such as NetFlow, IPFIX, and sFlow. But perhaps more useful, Open vSwitch supports a network state database (OVSDB) that supports remote triggers. Therefore, a piece of orchestration software can "watch" various aspects of the network and respond if/when they change. This is used heavily today, for example, to respond to and track VM migrations.

Open vSwitch also supports OpenFlow as a method of exporting remote access to control traffic. There are a number of uses for this including global network discovery through inspection of discovery or link-state traffic (e.g. LLDP, CDP, OSPF, etc.).

2.3.3 Maintenance of logical tags

Distributed virtual switches (such as VMware vDS and Cisco's Nexus 1000V) often maintain logical context within the network through appending or manipulating tags in network packets. This can be used to uniquely identify a VM (in a manner resistant to hardware spoofing), or to hold some other context that is only relevant in the logical domain. Much of the problem of building a distributed virtual switch is to efficiently and correctly manage these tags.

Open vSwitch includes multiple methods for specifying and maintaining tagging rules, all of which are accessible to a remote process for orchestration. Further, in many cases these tagging rules are stored in an optimized form so they don't have to be coupled with a heavyweight network device. This allows, for example, thousands of tagging or address remapping rules to be configured, changed, and migrated.

In a similar vein, Open vSwitch supports a GRE implementation that

can handle thousands of simultaneous GRE tunnels and supports remote configuration for tunnel creation, configuration, and tear-down. This, for example, can be used to connect private VM networks in different data centers.

2.3.4 Hardware Integration

Open vSwitch's forwarding path (the in-kernel datapath) is designed to be amenable to "offloading" packet processing to hardware chipsets, whether housed in a classic hardware switch chassis or in an end-host NIC. This allows for the Open vSwitch control path to be able to both control a pure software implementation or a hardware switch.

There are many ongoing efforts to port Open vSwitch to hardware chipsets. These include multiple merchant silicon chipsets (Broadcom and Marvell), as well as a number of vendor-specific platforms. The "Porting" section in the documentation discusses how one would go about making such a port.

The advantage of hardware integration is not only performance within virtualized environments. If physical switches also expose the Open vSwitch control abstractions, both bare-metal and virtualized hosting environments can be managed using the same mechanism for automated network control.

2.4 Summary

In many ways, Open vSwitch targets a different point in the design space than previous hypervisor networking stacks, focusing on the need for automated and dynamic network control in large-scale Linux-based virtualization environments.

The goal with Open vSwitch is to keep the in-kernel code as small as possible (as is necessary for performance) and to re-use existing subsystems when applicable (for example Open vSwitch uses the existing QoS stack). As of Linux 3.3, Open vSwitch is included as a part of the kernel and packaging for the userspace utilities are available on most popular distributions.

3 P4 Programming Language

3.1 Introduction

Before starting it's important to understand which programming language is used to build most of the devices used in the Intel Solution. P4 is a domain-specific programming language for network devices, specifying how data plane devices process packets. It is vendor-agnostic and protocol-agnostic and it aims at making easier the implementation of software packets processing pipeline instead of waiting years to develop a new chip. P4 is a declarative language, each P4 program follows the same workflow. P4 programs and compilers are target-specific (FPGA, Programmable ASICs, or software x86).

A P4 program classifies packets by header according to the definition provided by the programmer. As said before P4 is protocol-agnostic so you must declare every header you want to use because even the most common headers are not present in the standard libraries. The image below represents the working flow of a P4 program:

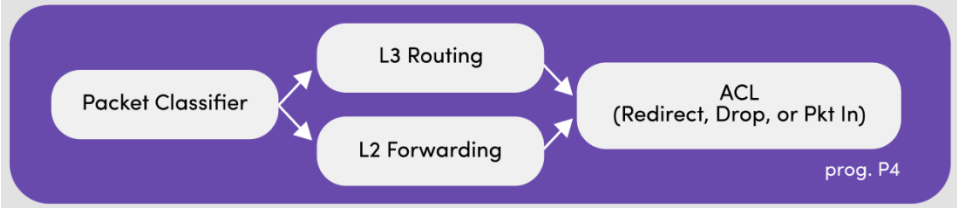

Figure 5: P4 Workflow

3.2 Motivation

The main motivation that is pushing us to program Data Planes instead of creating new chips is the very little time required to create a new program compared to the time needed to create a new chip. When you want to change functionality in your network you waste a lot of time developing a new chip and testing it while using a P4 programmable chip just have to change the code and re-compile. So what we should achieve is to create a chip to offload network functionalities in it instead of building chips having network functionalities wired in it.

3.3 Data Plane Declaration

P4 provides a single primitive type which is *bit* and it is used to define headers like the following one:

```
header ethernet_h {
    bit<48>   dst_addr;
    bit<48>   src_addr;
    bit<16>   ether_type;
}
```

Figure 6: Ethernet header defined in P4

In the source code, a classifier is called *Programmable Parser* and it is an ordered definition of headers that are used to match the inbound packets and classify them. A parser is defined as follows:

```
parser Ingress_Parser(
    packet_in pkt,
    out my_ingress_headers_t hdr,
    inout my_ingress_metadata_t meta,
    in psa_ingress_parser_input_metadata_t ig_intr_md,
    in empty_metadata_t resub_meta,
    in empty_metadata_t recirc_meta) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        pkt.extract(hdr.ethernet);
        transition select(hdr.ethernet.ether_type) {
            ETHERTYPE_IPV4:  parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        pkt.extract(hdr.ipv4);
        transition accept;
    }
}
```

Figure 7: Packet parser defined in P4

A parser is a state machine where each state works on a subset of the packet's fields and can fail or succeed. When a state fails an action must be performed, for example, drop the packet or forward it to another state. Once the parsing is done there is a pipeline of *Match-Action* called *Programmable Match-Action Pipeline* and a valid packet must be processed by each step of the pipeline. An action of the pipeline is used to change header values, decide the destination, or change addresses. In the following example there is a single step that defines IPv4 forwarding:

```
control ingress(
    inout my_ingress_headers_t hdr,
    inout my_ingress_metadata_t meta,
    in psa_ingress_input_metadata_t ig_intr_md,
    inout psa_ingress_output_metadata_t ostd) {

    action send(PortId_t port) {
        ostd.egress_port = (PortId_t) port;
    }

    action drop() {
        ostd.drop = true;
    }

    table ipv4_host {
        key = { hdr.ipv4.dst_addr : exact; }
        actions = {
            send;drop;
            @defaultonly NoAction;
        }

        const default_action = drop();

        size = IPV4_HOST_SIZE;
    }

    apply {
                ipv4_host.apply();
    }
}
```

Figure 8: Packet ingress defined in P4

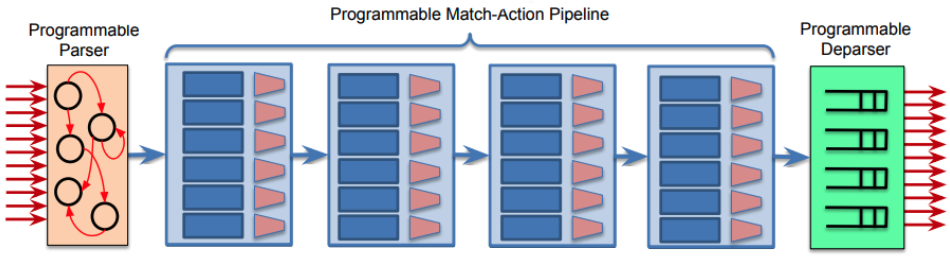

Figure 9: Programmable Pipeline

The last piece of the pipeline is the *Programmable Deparser* which decides how the packets will look in the wire. It emits headers in the order that they should be transmitted in the wire. The syntax used to define a de-parser is the same as shown for the parser[4].

```
control MyDeparser(packet_out packet, in headers hdr) {
  apply {
    packet.emit(hdr.ethernet);
    packet.emit(hdr.ipv4);
  }
}
```

In the context of IPDK, P4 is used to define how the integrated OVS (Open vSwitch) should process packets.

3.4 P4 and eBPF

There are a few differences between P4 and eBPF but what is important to understand is that lately, the community is trying to make them compatible. There are a lot of good projects that try to generate eBPF code from P4 and vice-versa. The main differences are listed below:

- eBPF is a general purpose framework while P4 is a specific purpose framework. eBPF works on bytecode and registers while P4 works on headers parser and match-action tables.
- The safety is ensured by the kernel in eBPF while is ensured by design in P4.

Some good projects that are working on the translation between P4 and eBPF are:

- p4c-ebpf
- p4c-xdp
- p4c-ubpf

But some of us could ask why we are trying to write P4 and translate it in eBPF, or vice-versa. There is more than one motivation:

- P4 is less flexible than eBPF, which can make us think that this is a disadvantage but when it comes to certain kinds of formal correctness analysis, a more restricted language can make such tasks easier to do.
- Writing P4 is more secure since is a declarative language so it is not affected by most of the programming errors that are possible in C-like languages used to write eBPF.

Is important to note that we said that P4 is less flexible than eBPF but this assertion is true when we go deep into the usage and we want to do something very specific but in the general usage both are very similar to each other.

4 IPDK

4.1 Introduction

IPDK is an open-source development framework, community-driven and target-agnostic which runs on CPU, IPU, DPU or switch. The main maintainer is Intel but also other companies are implied like Marvell, Ericsson, etc. What IPDK does is create an abstraction level between the target and the implementation to offload and manage infrastructures that run on different targets. Under the hood, it uses a few well-established linux frameworks like SPDK, DPDK and P4.

IPDK introduces two different standardized interfaces:

- **Infrastructure Application Interface**: Developed starting from pre-existing networking interfaces. It uses RPC mechanisms to allow applications to run locally, remotely or a mix of both. This interface uses a couple of tools:
  - **P4Runtime**: Used for programmable networking data plane
  - **OpenConfig**: Used for configuring physical ports, virtual devices, QoS and inline operations such as IPsec
- **Target Abstraction Interface**: The Target Abstraction Interface (TAI) is an abstraction exposed by an infrastructure device running infrastructure applications that serve connected compute instances (attached hosts and/or VMs, which may or may not be containerized). The infrastructure device may also contain an isolated Infrastructure Manager used to help manage the lifecycle of the device.

In the image below it's possible to see how the two interfaces work and in which kind of communication they are involved[5].

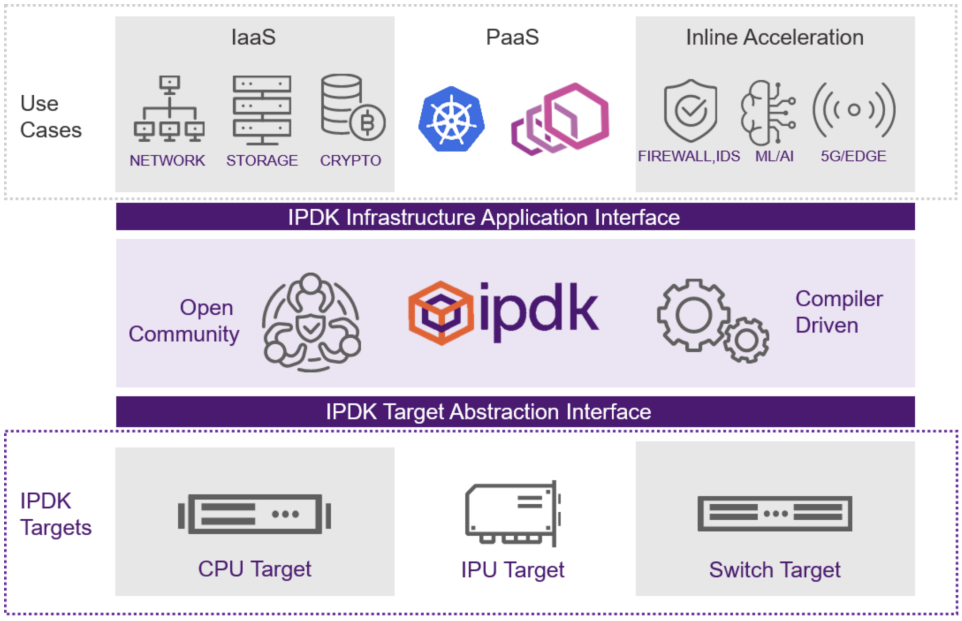

Figure 10: IPDK layers and interfaces interactions

Using IPDK it is possible to build either Virtual Networking or Virtual Storage environments.

4.1.1 Virtual Networking

When we talk about VN in the context of IPDK we are referring to a single virtual device type which is *virtio-net*. Is it possible to create a Virtual Networking in a bare metal server that hosts VMs or in a nested virtualized environment, in both cases the configuration is the same.

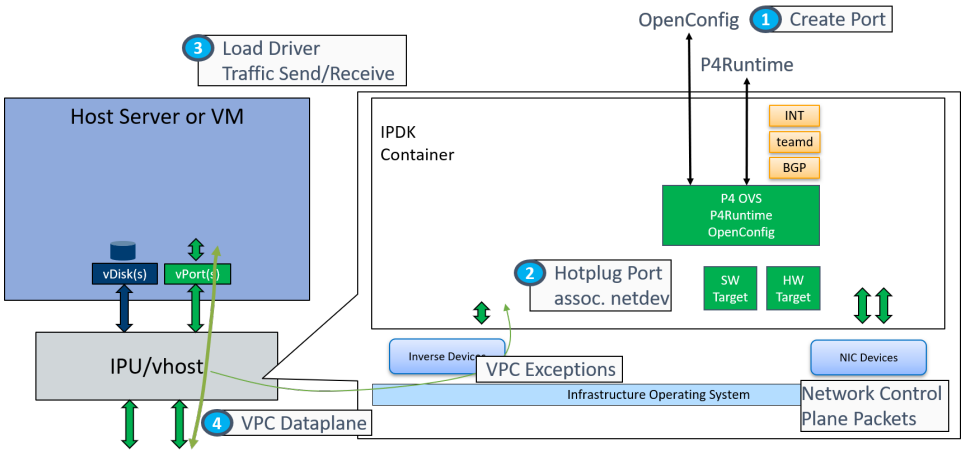

Figure 11: Virtual Networking Example

In the image above we can see 4 different steps needed to hotplug a new virtual device, in this case, SW Target

- **1.** Over OpenConfig, create a new virtual port (specifying the type, number of queues, etc.).
- **2.** This new virtual port is associated with a netdev in P4 OVS, and a corresponding port device is hotplugged into the host or VM. Any exception traffic from that hotplug'd device will arrive on this netdev. Any traffic P4 OVS wants to direct to this device can be sent over this netdev as well.
- **3.** The tenant instance loads the corresponding driver for that device and can now send/receive traffic.
- **4.** The virtual private cloud data-plane moves traffic to/from the instance, sending exceptions to the infrastructure software when needed[6].

4.1.2 Virtual Storage

In this case, there are 3 different devices supported by the suite:

- virtio-blk
- virtio-scsi

- NVMe

Using the same schema that we've seen in the VN part we can describe how the Virtual Storage management is achieved

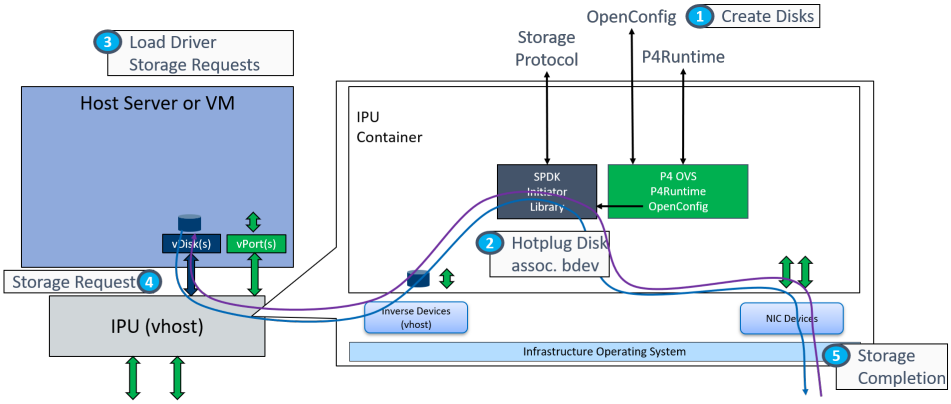

Figure 12: Virtual Storage Example

- **1.** Over OpenConfig, create a new virtual storage controller device, specifying the device type, number of queues, etc.
- **2.** The storage controller has one or more attached volumes. The maximum number of volumes is dependent on the device type. Each volume is associated with an SPDK block device (bdev) that implements the actual storage backing the volume.
- **3.** The tenant instance loads the corresponding driver for that device, and can then send storage requests to volumes attached to the storage controller that will be processed by the SPDK storage application.
- **4.** Storage requests (reads, writes, flushes) directed to a volume will arrive at the SPDK target layer and be translated and forwarded to its associated SPDK bdev. SPDK bdevs can be of many different types, depending on the type of backing storage for the volume. Examples include NVMe over TCP, NVMe over RDMA, or Ceph RBD.
- **5.** The SPDK bdev module responsible for a given bdev type forwards the storage request to the bdev's associated remote target. For

example, the NVMe/TCP bdev module is responsible for establishing TCP connections to the remote NVMe/TCP target and sending NVMe CONNECT commands to associate those connections with the NVMe subsystem containing the volume associated with the tenant's virtual storage controller. It also constructs NVMe commands for the storage request(s) and sends them (along with any write data) to the remote target over the previously established TCP socket.

IPDK provides a good number of recipes where to start to develop a custom solution. Almost all the solutions are based on P4-OVS and a P4 program.

4.2 Build, Deployment and Management

The most interesting thing about IPDK is that it introduces some management components that are dynamically created during the build process based on the P4 program. To better understand what happens under the hood let's see some main commands provided by IPDK that generate the management layer.

P4 Compile

```
export OUTPUT_DIR=/root/ipdk-io/build/networking/examples/simple_l3
mkdir $OUTPUT_DIR/pipe
p4c-dpdk --arch pna --target dpdk \
    --p4runtime-files $OUTPUT_DIR/p4Info.txt \
    --bf-rt-schema $OUTPUT_DIR/bf-rt.json \
    --context $OUTPUT_DIR/pipe/context.json \
    -o $OUTPUT_DIR/pipe/simple_l3.spec $OUTPUT_DIR/simple_l3.p4
```

Figure 13: P4 compile command

the command above compiles the P4 program and it creates 3 different files: **bf-rt.json**, **context.json** and **p4info.txt** which are used in the next command to generate a management layer.

Build Pipeline Management Layer

```
./install/bin/tdi_pipeline_builder \
    --p4c_conf_file=$OUTPUT_DIR/simple_l3.conf \
    --bf_pipeline_config_binary_file=$OUTPUT_DIR/simple_l3.pb.bin
```

Figure 14: Generate a management layer based on the compilation output

the command above takes **.conf** file which contains the locations of the 3 files generated by the previous command. This command combines the artifacts generated by p4c compiler to generate a single bin file (*.pb.bin) to be pushed to the target.

Management

When a deployment is performed by IPDK we will find a management layer that helps us to modify the target configuration. This management layer is a gRPC server running on the target that can be queried by some CLIs such as:

- **ovs-p4rt**: A library (C++ with a C interface) that allows ovs-vswitchd and ovsdb-server to communicate with the P4Runtime Server in infrap4d via gRPC. It is used to program (insert/modify/delete) P4 forwarding tables in the pipeline.
- **p4rt-ctl**: A Python-based P4Runtime client which talks to the P4Runtime Server in infrap4d via gRPC, to program the P4 pipeline and insert/delete P4 table entries.
- **gnmi_cli**: A gRPC-based C++ network management interface client to handle port configurations and program fixed functions in the P4 pipeline.

Those clients above communicate to a runtime deployed by IPDK into the target. The main component is called infrap4d described as follows:

4.2.1 Infrap4d

Infrap4d integrates Stratum, the Kernel Monitor (krnlmon), Switch Abstraction Interface (SAI), Table Driven Interface (TDI), and a P4 target driver into a separate process (daemon).

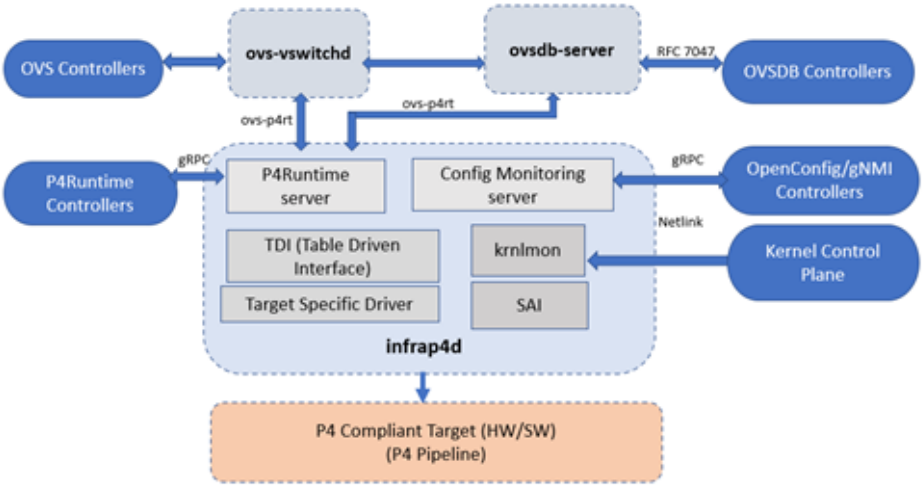

Figure 15: Infrap4d process schema

In the previous image the target used as example is P4-OVS. Infrap4d provides 2 interfaces:

- **P4Runtime**: The P4Runtime API is a control plane specification for managing the data plane elements of a device defined or described by a P4 program.
- **gNMI**: gRPC Network Management Interface (gNMI) is a gRPC-based protocol to manage network devices.

Stratum

Stratum is an open-source silicon-independent switch operating system. It is a component of Infrap4d that provides the P4Runtime and gNMI/Openconfig capabilities for P4 flow rule offloads and port configuration offloads. Stratum is augmented with a new tdi platform layer that processes P4rt and

gNMI requests and interacts with the underlying P4 target driver through TDI. A new ipdk platform layer provides IPDK-specific replacements for several TDI modules that allow it to handle configuration differences between IPUs and the switches for which Stratum was developed.

TDI - Table Driven Interface

TDI (Table Driven Interface) provides a target-agnostic interface to the driver for a P4-programmable device. It is a set of APIs that enable configuration and management of P4 programmable and fixed functions of a backend device in a uniform and dynamic way. Different targets like bmv2 and P4-DPDK can choose to implement their own backends for different P4 and non-P4 objects but can share a common TDI. Stratum talks to the target-specific driver through the TDI front-end interface.

krnlmon - Kernel Monitor

The Kernel Monitor receives RFC 3549 messages from the Linux Kernel over a Netlink socket when changes are made to the kernel networking data structures. It listens for network events (link, address, neighbor, route, tunnel, etc.) and issues calls to update the P4 tables via SAI and TDI. The kernel monitor is an optional component of infrap4d.

SAI - Switch Abstraction Interface

Switch Abstraction Interface (SAI) defines a vendor-independent interface for switching ASICs.

4.3 L3 Forwarding Example

Prepare the environment

Using a new folder called SDE as root folder, assuming /root/SDE for the rest of the example, run the following commands

```
git clone --recursive https://github.com/p4lang/p4-dpdk-target.git
```

Figure 16: Cloning the DPDK target

```
sudo -s
mkdir install
```

Figure 17: Create an install folder to store the executable, deps and libs

```
cd p4-dpdk-target/tools/setup
source p4sde_env_setup.sh /root/SDE
```

Figure 18: Source envs

```
pip3 install distro (dependency)
cd p4-dpdk-target/tools/setup
python3 install_dep.py
```

Figure 19: Install dependencies that will be stored in /root/SDE/install

```
cd $SDE/p4-dpdk-target
git submodule update --init --recursive --force
./autogen.sh
./configure --prefix=$SDE_INSTALL
make -j
make install
```

Figure 20: Install P4 DPDK Target

Now P4 DPDK Target is installed and ready to be configured so we can install the dependencies and build the P4 program. Let's clone the *Networking Recipe* from *https://github.com/ipdk-io/networking-recipe*. Install some dependencies by running *apt install libatomic1 libnl-route-3-dev* and *pip3 install -r requirements.txt* from inside the networking-recipe folder.

Install Infrap4d

Run all the following commands from inside the networking-recipe folder.

```
git clone --recursive https://github.com/ipdk-io/networking-recipe.git ipdk.recipe
cd ipdk.recipe
export IPDK_RECIPE=`pwd`
cd $IPDK_RECIPE/setup
cmake -B build -DCMAKE_INSTALL_PREFIX=<dependency install path> [-DUSE_SUDO=ON]
cmake --build build [-j<njobs>]
```

Figure 21: Install Infrap4d

Export 2 variables:

- DEPEND_INSTALL=folder where to store deps, i.e. /root/deps
- SDE_INSTALL=/root/SDE/install

After being sure that the exported variables are correct run the following command

```
source ./scripts/dpdk/setup_env.sh $IPDK_RECIPE $SDE_INSTALL $DEPEND_INSTALL
```

Figure 22: Exports envs required for the setup proces

Now we can finally compile the network recipe. This step may take a lot of time because involves source compilation, components building and downloading:

```
cd $IPDK_RECIPE
./make-all.sh --target=dpdk
```

Figure 23: Compile the recipe

Run Infrap4d

The following commands will run the Infrap4d daemon, run it from inside networking-recipe folder.

```
source ./scripts/dpdk/setup_env.sh $IPDK_RECIPE $SDE_INSTALL $DEPEND_INSTALL
sudo ./scripts/dpdk/copy_config_files.sh $IPDK_RECIPE $SDE_INSTALL
sudo ./scripts/dpdk/set_hugepages.sh
sudo ./install/sbin/infrap4d
```

Figure 24: Enable hugepages and run Infrap4d

Run Network Recipe

Create 2 ports using the gnmi-ctl, run it from inside networking-recipe folder:

```
sudo ./install/bin/gnmi-ctl set "device:virtual-device,name:TAP0,pipeline-name:pipe,mempool-name:MEMPOOL0,mtu:1500,port-type:TAP"
sudo ./install/bin/gnmi-ctl set "device:virtual-device,name:TAP1,pipeline-name:pipe,mempool-name:MEMPOOL0,mtu:1500,port-type:TAP"
ifconfig TAP0 up
ifconfig TAP1 up
```

Figure 25: Create 2 different TAP ports

clone the P4 program https://github.com/ipdk-io/ipdk.git. Compile the P4 example program:

```
export OUTPUT_DIR=/root/ipdk-io/build/networking/examples/simple_l3
mkdir $OUTPUT_DIR/pipe
p4c-dpdk --arch pna --target dpdk \
    --p4runtime-files $OUTPUT_DIR/p4Info.txt \
    --bf-rt-schema $OUTPUT_DIR/bf-rt.json \
    --context $OUTPUT_DIR/pipe/context.json \
    -o $OUTPUT_DIR/pipe/simple_l3.spec $OUTPUT_DIR/simple_l3.p4
```

Figure 26: Compile the P4 Program

```
./install/bin/tdi_pipeline_builder \
    --p4c_conf_file=$OUTPUT_DIR/simple_l3.conf \
    --bf_pipeline_config_binary_file=$OUTPUT_DIR/simple_l3.pb.bin
```

Figure 27: Build the pipeline

Now that the pipeline is ready we can set it as current pipeline of the p4runtime which is running in the infrap4d process:

```
sudo ./install/bin/p4rt-ctl set-pipe br0 $OUTPUT_DIR/simple_l3.pb.bin $OUTPUT_DIR/p4Info.txt
```

Figure 28: Set the pipeline

We can add an arbitrary number of rules to this pipe but for the moment let's add two rules:

```
sudo  ./install/bin/p4rt-ctl add-entry br0 ingress.ipv4_host "hdr.ipv4.dst_addr=1.1.1.1,action=ingress.send(0)"
sudo  ./install/bin/p4rt-ctl add-entry br0 ingress.ipv4_host "hdr.ipv4.dst_addr=2.2.2.2,action=ingress.send(1)"
```

Figure 29: Set pipe's rules

In the commands above we have a bunch of parameters explained below:

- **br0**: the name of the vSwitch assigned internally by p4runtime
- **ingress.ipv4_host**: the name of the table in *dot notation* where the rule should be added
- **hdr.ipv4.dst_addr=<ip>,action=<action>(<params>)**: is a string which explains what the entry should do. You should read it as following: when the destination ip of the incoming packet, which is the key of the table, is equal to <ip> perform the action called <action> having <params> as parameters.

To test the solution we can simply use a python library called Scapy. To use it run Python3 in a terminal session, and import everything of scapy ***from scapy.all import ** and send a packet with the following instruction:

```
eth = Ether(dst="00:00:00:00:03:14", src="a6:c0:aa:27:c8:2b")
ip = IP(src="192.168.1.10", dst="2.2.2.2")
udp = UDP()
raw = Raw(load="0"*50)
sendp(eth/ip/udp/raw, iface='TAP0')
```

Figure 30: Send packet using python CLI

You can use TCPDUMP to sniff packets on the TAP1 interface and see that the packet has been forwarded correctly. You can try to remove the forwarding rule and you'll see that the packets are not forwarded anymore from TAP0 to TAP1. To remove the rule run

./install/bin/p4rt-ctl del-entry br0 ingress.ipv4_host
"hdr.ipv4.dst_addr=2.2.2.2"

5 Kubernetes

5.1 Support

Is there a repository that aims to develop a recipe to use IPDK in a kubernetes environment. This recipe has some constraints that make difficult its usability in a real-world use case. The recipe is on GitHub, at the link https://github.com/ipdk-io/k8s-infra-offload. At the moment this recipe is working only with Open vSwitch but in the future will be possible to use it also with hardware target.

There is a big problem related to hardware targets: a large amount of local packets will be processed by the SmartNic once the recipe is offloaded to it. This incredibly high number of packets may be a problem from a performance point of view.

Acronyms

DPDK Data Plane Development Kit. 18

IPDK Infrastructure Programmer Development Kit. 18, 19, 22

P4 P4 Programming Language. 18

RPC Remote Procedure Call. 18

SPDK Storage Performance Development Kit. 18

SW Software. 20

VM Virtual Machine. 19

VN Virtual Networking. 19, 21

VS Virtual Storage. 21

References

[1] Evaluation Engineering, SmartNIC Architectures: A Shift to Accelerators and Why FPGAs are Poised to Dominate, 18 Oct, 2020

[2] P4 Open vSwitch, https://ipdk.io/documentation/

[3] IPDK Targets, https://ipdk.io/documentation/targets

[4] P4 Docs, https://p4.org/p4-spec/docs/P4-16-v-1.2.3.html

[5] IPDK Interfaces, https://shorturl.at/ajNX7

[6] IPDK Virtual Networking, https://shorturl.at/pszCE

[7] IPDK Virtual Storage, https://shorturl.at/hjMN1