Содержание

DPDK Interfaces Configuration

DPDK (Data Plane Development Kit) allows working with network cards directly without actually using the Linux kernel. This improves the performance of the solution. DPDK supports many more models of network cards than pf_ring, and a much richer interface. So it allows you to implement various working schemes, suitable for 10G, 25G, 40G, 100G traffic, etc.

System Preparation

The initial installation of DPI is done by VAS Experts technical support. Please do not try to do the initial installation yourself, as we may need to check all the steps you have done later, which increases the workload of tech support.

Later on you will be able to add or remove network ports and change the configuration yourself.

Ports configuration

The network cards that Stingray will work with are removed from the control of the operating system and therefore are not visible as Ethernet devices to the operating system. The DPDK addresses Ethernet devices by their PCI identifiers, which can be obtained by a command:

```
lspci -D|grep Eth

0000:04:00.0 Ethernet controller: Intel Corporation 82599ES 10-Gigabit
SFI/SFP+ Network Connection (rev 01)
0000:04:00.1 Ethernet controller: Intel Corporation 82599ES 10-Gigabit
SFI/SFP+ Network Connection (rev 01)
```

This command outputs a list of all ethernet-type PCI devices. Each line starts with a PCI device system identifier – these PCI identifiers are the unique identifiers of the network card in the DPDK.

The list of cards in DPDK mode can be checked with the command:

```
driverctl list-overrides

0000:04:00.0 vfio-pci
0000:04:00.1 vfio-pci
```

If necessary, the cards can be taken out of DPDK mode with a command and the regular Linux driver is activated for them.

You will need to stop the Fastdpi process beforehand.

```
service fastdpi stop
```

```
driverctl unset-override 0000:04:00.0
driverctl unset-override 0000:04:00.1
```

After working with the regular driver, do not forget to put them back under DPDK control with the command:

```
driverctl -v set-override 0000:04:00.0 vfio-pci
driverctl -v set-override 0000:04:00.1 vfio-pci
```

When switching cards into DPDK mode be careful not to accidentally switch the server control interface into DPDK mode – communication with the server will be immediately cut off!

In older installations, the igb_uio driver was used instead of vfio-pci, as you can see in the output of the command

```
driverctl list-overrides
0000:04:00.0 igb_uio
```

In this case it is recommended to switch to the vfio-pci driver. To do this, run these commands for all devices in the list of list-overrides:

```
echo "options vfio enable_unsafe_noiommu_mode=1" >
/etc/modprobe.d/vfio-noiommu.conf
driverctl -v set-override 0000:04:00.0 vfio-pci
```

Setting `enable_unsafe_noiommu_mode=1` may require a server reboot.

Stingray SG Configuration

When the system is configured to work with DPDK, you can start configuring the Stingray SG. The interfaces are configured with «in»-«out» pairs (for the future convenience, the «in» interface should face the operator's internal network, and the "out" - the uplink). Each pair forms a network bridge that is L2 transparent. PCI identifiers are used as interface names with the replacement of ':' by '-' (because the symbol ':' in the interface name is reserved in Stingray SG to separate interfaces in one cluster):

```
   # In - port 41:00.0
in_dev=41-00.0
   # Out - port 41:00.1
out_dev=41-00.1
```

This configuration sets a single bridge 41-00.0 ←→ 41-00.1
You can specify a group of interfaces with ':'

```
in_dev=41-00.0:01-00.0:05-00.0
out_dev=41-00.1:01-00.1:05-00.1
```

This group forms the following pairs (bridges):
41-00.0 ←→ 41-00.1
01-00.0 ←→ 01-00.1
05-00.0 ←→ 05-00.1
The pairs must have devices of the same speed; it is unacceptable to pair 10G and 40G cards. However, the group can have interfaces of different speeds, for example, one pair is 10G, the other is 40G.

The maximum ethernet packet size on the devices is set by the `snaplen` option in fastdpi.conf, by default `snaplen=1540`.

Setting device aliases

Starting from version 9.5.3, the SSG now allows you to specify aliases for devices. This is due to the fact that DPDK supports numerous devices, not only PCI devices, but also, for example, vmbus devices (Hyper-V) or virtual (vdev) devices. Additionally, each DPDK driver supports its own set of configuration parameters for fine-tuning. The syntax of describing such devices is incompatible with the syntax of the `in_dev'/'out_dev` task, so the notion of an alias device has been introduced.

The essence of the alias is very simple: you describe the desired device in a separate parameter and give this description a name. Then in the `in_dev`, `out_dev`, `tap_dev` (and in all other parameters that refer to devices from `in_dev` and `out_dev`) you specify this name – the alias of the device.

Each alias is specified by a separate `dpdk_device` parameter:

```
dpdk_device=alias:bus:device-description
```

Here:

- `alias` specifies an alias of the device (e.g. eth1). Only letters and numbers are allowed in the alias.
- `bus` – bus type: `pci`, `vmbus`, `vdev`.
- `device-description` – device descriptor in DPDK syntax.

For example:

```
  # eth1 is the alias of PCI device 41:00.0
dpdk_device=eth1:pci:41:00.0
  # eth2 eth2 is the alias of PCI device 41:00.1
dpdk_device=eth2:pci:41:00.1

in_dev=eth1
out_dev=eth2
```

This description is equivalent to the following:

```
in_dev=41-00.0
out_dev=41-00.1
```

Note that in `dpdk_device` the PCI device is specified in the canonical form `41:00.0`.

For PCI devices, assigning to `in_dev`/`out_dev` via aliases is not necessary, you can use the old notation.

If you want to connect Hyper-V devices (and these are not PCI devices, but VMbus devices), then the use of aliases is mandatory. Example:

```
dpdk_device=subs1:vmbus:392b7b0f-dbd7-4225-a43f-4c926fc87e39
dpdk_device=subs2:vmbus:58f75a6d-d949-4320-99e1-a2a2576d581c,latency=30
dpdk_device=inet1:vmbus:34f1cc16-4b3f-4d8a-b567-a0eb61dc2b78
dpdk_device=inet2:vmbus:aed6f53e-17ec-43f9-b729-f4a238c49ca9,latency=30
in_dev=subs1:subs2
out_dev=inet1:inet2
```

Here we not only set the alias, but also specify the `latency=30` argument for the DPDK driver. In fact, each DPDK driver supports its own set of arguments, see DPDK documentation for the corresponding version (The version of the DPDK from which the SSG is built is displayed in the `fastdpi_alert.log` at startup, as well as when calling `fastdpi -ve`). It should be noted that careless setting of arguments for the driver can lead to hard-to-detect errors and SSG performance downgrade, so do not use this feature without consulting our technical support.

Configuration in Hyper-V

Starting with version 9.5.3, SSG supports running in a Hyper-V virtual machine. On guest VEOS 8.6 must be installed:

```
   # Multi-queue support - required for SSG
dnf install kernel-modules-extra
```

Host system (Windows) must support multiple channels for virtualized NICs

Devices on Hyper-V are VMBus, and not PCI devices, so they require a special conversion to DPDK mode. Each device (interface) is identified by its unique UUID, so first you need to know the UUIDs of all interfaces that SSG will work with. Then you have to put the device into DPDK mode:

```
# switch the interfaces eth0 and eth2 into DPDK mode
for DEV in eth0 eth2
do
    # get the UUID for the device
    DEV_UUID=$(basename $(readlink /sys/class/net/$DEV/device))
    # switch to DPDK compatible mode
```

```
    driverctl -b vmbus set-override $DEV_UUID uio_hv_generic

    # Device appears in
    # /sys/bus/vmbus/drivers/uio_hv_generic/$DEV_UUID

    echo "$DEV uuid=$DEV_UUID"
done
```

If necessary, the interface can be switched back to kernel-mode like this:

```
ETH0_UUID=<eth0_UUID>
driverctl -b vmbus unset-override $ETH0_UUID
```

Next, configure the SSG – set the devices in `fastdpi.conf`. While doing so, use alias to specify the UUIDs that we have just learned:

```
   # eth0 UUID=392b7b0f-dbd7-4225-a43f-4c926fc87e39
dpdk_device=eth0:vmbus:392b7b0f-dbd7-4225-a43f-4c926fc87e39
   # eth2 UUID=34f1cc16-4b3f-4d8a-b567-a0eb61dc2b78
dpdk_device=eth2:vmbus:34f1cc16-4b3f-4d8a-b567-a0eb61dc2b78

   # then use the aliases eth0 and eth2 everywhere when specifying the
devices
in_dev=eth0
out_dev=eth2
```

Clusters

The DPDK version of Stingray SG supports clustering: you can specify which interfaces are included in each cluster. The clusters are separated with the '|' symbol.

```
in_dev=41-00.0|01-00.0:05-00.0
out_dev=41-00.1|01-00.1:05-00.1
```

This example creates two clusters:

- cluster with bridge 41-00.0 ←→ 41-00.1
- cluster with bridges 01-00.0 ←→ 01-00.1 and 05-00.0 ←→ 05-00.1

Clusters are a kind of a legacy of the Stingray SG pf_ring-version: in pf_ring, cluster is the basic concept of "one dispatcher thread + RSS handler threads" and is almost the only way to scale. The disadvantage of the cluster approach is that the clusters are physically isolated from each other: it is impossible to forward a packet from the X-interface of cluster #1 to the Y-interface of cluster #2. This can be a significant obstacle in the SKAT L2 BRAS mode.

In DPDK, clusters are also isolated from each other, but unlike pf_ring, here a cluster is a more logical concept inherited from pf_ring. DPDK is much more flexible than pf_ring and allows you to build complex multi-bridge configurations with many dispatchers without using clusters. In fact, the only "pro" argument for clustering in the Stingray-DPDK version is the case when you have two

independent networks A and B connected to the Stingray SG, which should not interact with each other in any way.

Tip: instead of using clusters, consider switching to a different `dpdk_engine`, that is more suitable for your load.

The following descriptions of configurations assume that there is only one cluster (no clustering).

Number of Cores (Threads)

CPU cores are perhaps the most critical resource for the Stingray SG. The more physical cores there are in the system, the more traffic can be processed by the SSG.

Stingray SG does not use Hyper-Threading: only real physical cores are taken into account, not logical ones.

Stingray SG needs the following threads to operate:

- processing threads - process incoming packets and write to the TX-queue of the card;
- dispatcher threads - read the card's RX queues and distribute incoming packets among processing threads;
- service threads - perform deferred (time-consuming) actions, receive and process fdpi_ctrl and CLI, connection with PCRF, sending netflow
- system kernel - dedicated to the operating system.

Processing and dispatcher threads cannot be located on the same core. At start, Stingray SG binds threads to cores. Stingray SG by default selects the number of handler threads depending on the interface speed:
10G - 4 threads
25G - 8 threads
40G, 50G, 56G - 16 threads
100G - 32 threads
For a group, the number of threads is equal to the sum of threads number for each pair; e.g., for the cards:

```
   # 41-00.x - 25G NIC
   # 01-00.x - 10G NIC
in_dev=41-00.0:01-00.0
out_dev=41-00.1:01-00.1
```

12 processing threads will be created (8 for 25G card and 4 for 10G card)

In fastdpi.conf, you can specify the number of threads per bridge using the `num_threads` parameter:

```
   # 41-00.x - 25G NIC
   # 01-00.x - 10G NIC
```

```
in_dev=41-00.0:01-00.0
out_dev=41-00.1:01-00.1

num_threads=4
```

This configuration will create 8 (num_threads=4 * 2 bridges) processing threads.

Stingray SG, when planning cores, takes into account the NUMA node, which includes the cores and the card: if the card is on NUMA node 0, the SSG will assign handler threads and dispatcher threads to NUMA node 0 as well. If there are not enough cores in the NUMA node, the SSG will not start.

In addition to the handler threads, for operating you also need at least one dispatcher thread (and therefore at least one more core) that reads the rx-queues of the interfaces. The dispatcher's task is to ensure that packets belonging to the same flow get into the same handler flow.

The internal architecture of working with one or many dispatchers is strikingly different, therefore Stingray provides several engines configured by the `dpdk_engine` parameter of the fastdpi.conf file:

- `dpdk_engine=0` - read/write default engine, one dispatcher for all;
- `dpdk_engine=1` - read/write engine with two dispatcher threads: for each direction by dispatcher;
- `dpdk_engine=2` - read/write engine with RSS support: for each direction `dpdk_rss` dispatchers are created (`dpdk_rss=2` by default). Thus, the total number of dispatchers = 2 * `dpdk_rss`;
- `dpdk_engine=3` - read/write engine with a separate dispatcher for each bridge.

Further, all these engines are described in detail, their configuration features and areas of application, and the dispatcher threads in gneral.

Explicit Binding to Cores

You can explicitly bind threads to cores in fastdpi.conf. The parameters:

- `engine_bind_cores` - list of core numbers for processing threads
- `rx_bind_core` - list of core numbers for dispatcher threads.

The format for specifying these lists is the same:

```
# 10G cards - 4 processor threads, 1 dispatcher per cluster
in_dev=01-00.0|02-00.0
out_dev=01-00.1|02-00.1

# Bind processing threads for cluster #1 to cores 2-5, dispatcher to core 1
#    for cluster #2 - to cores 7-10, dispatcher to core 6
engine_bind_cores=2:3:4:5|7:8:9:10
rx_bind_core=1|6
```

Without clustering:

```
# 10G cards - 4 processing threads per card
in_dev=01-00.0:02-00.0
out_dev=01-00.1:02-00.1
# 2 dispatchers (by directions)
dpdk_engine=1

# Bind processing threads and dispatcher threads
engine_bind_cores=3:4:5:6:7:8:9:10
rx_bind_core=1:2
```

As noted, the handler and dispatcher threads must have dedicated cores; it is not allowed to bind several threads to one core - the Stingray SG will display an error in fastdpi_alert.log and will not start.

Explicit binding to cores can only be applied in emergency cases; automatic binding is usually enough. To find out the core numbers, we advise you to run the SSG with automatic binding (without `engine_bind_cores` and `rx_bind_core` parameters) and look at the dump of the system topology in fastdpi_alert.log: core number is lcore

With the explicit binding, SSG strictly follows the parameters specified in fastdpi.conf and does not take into account the NUMA node, which may negatively affect performance (minus 10% - 20%)

The Dispatcher Thread Load

If the load of the dispatcher thread is close to 100%, it does not mean that the dispatcher cannot cope: DPDK assumes that data from the card is read by the consumer (this is the dispatcher) without any interruptions, so the dispatcher constantly queries the state of interfaces rx-queues for the presence of packets (the so-called poll mode). If no packet is received within N polling cycles, the dispatcher is disabled for a few microseconds, which is quite enough to reduce the load on the core to several percent. But if packets arrive once in N-i polling cycles, the dispatcher will not enter the sleep mode and the core will be loaded at 100%. This is normal.

The load of SSG threads can be viewed with the following command:

```
top -H -p `pidof fastdpi`
```

The real state of each dispatcher can be seen in fastdpi_stat.log, – it also displays statistics on dispatchers in the following form:

```
[STAT    ][2020/06/15-18:17:17:479843]  [HAL][DPDK] Dispatcher statistics
abs/delta:
                    drop (worker queue full)           |  empty NIC RX |
RX packets
```

```
    Cluster #0:                0/0           0.0%/  0.0% |   98.0%/95.0% |
100500000/100500
```

here `empty NIC RX` - this is the percentage of empty polls of cards rx-queues - an absolute percentage (since the beginning of the Stingray SG operation) and relative (delta since the last output in the stat-log). 100% means that there are no input packets, the dispatcher is idle. If the relative percentage is less than 10 (that is, in more than 90% of interface polls there are ingoing packets), the dispatcher cannot cope and it is necessary to consider another engine with more dispatchers.

There is another good indicator that the current engine cannot cope: a non-zero delta value for the `drop (worker queue full)`. This is the number of dropped packets that the dispatcher was unable to send to the processing thread because the processor's input queue was full. This means that the handlers are unable to handle incoming traffic. This can happen because of two reasons:

- either there are too few processing threads, you need to increase the `num_threads` parameter or choose another engine (the `dpdk_engine` parameter);
- or the traffic is heavily skewed and most of the packets go to one or two handlers, while the rest are free. In this situation, you need to analyze the traffic structure. You can try to increase or decrease the number of handler threads by one, so that the dispatcher hash function would distribute packets more evenly (the number of the processing thread is `hash_package mod number_of_handlers`).

dpdk_engine=0: One dispatcher

In this mode, Stingray SG creates one dispatcher thread per cluster. The dispatcher reads incoming packets from all in_dev and out_dev devices and distributes the packets to the handler threads. Suitable for 10G cards, withstands loads up to 20G or more (depends on the CPU model and the check_tunnels parsing mode)

The total number of cores required is equal to the number of handlers plus one core per dispatcher.

Stingray SG configures cards as follows:

- RX queue count = 1
- TX queue count = number of processing threads. Processing threads record data directly each to their TX-card queue.

For read-only mode (without `out_dev`) the TX queue number is zero. Some DPDK drivers (e.g. vmxnet3) do not allow to configure the card with TX queue number equal to zero. For such drivers, in SSG version 10.2 the fastdpi.conf parameter dpdk_txq_count is introduced: `dpdk_txq_count=1`

dpdk_engine=1: Dispatchers by direction

In this mode, two dispatcher threads are created: one for directing from subscribers to inet (for

in_dev), the other for directing from inet to subscribers (for out_dev). Suitable for loads over 20G (25G, 40G cards).

The total number of cores required is equal to the number of handlers plus two cores per dispatcher.

Stingray SG configures cards as follows:

- RX queue count = 1
- TX queue count = number of processing threads. Processing threads record data directly each to their TX-card queue.

dpdk_engine=2: RSS support

In this mode, RSS (receive side scaling) cards are used. The RSS value is set in fastdpi.conf with the parameter:

```
dpdk_rss=2
```

The `dpdk_rss` value must not be less than 2. For each direction, dispatcher `dpdk_rss` is created.

The total number of cores required is equal to the number of handlers plus `dpdk_rss` * 2 per dispatchers

Suitable for powerful 50G+ cards (for SSG-100+). If you have a grouping of 50G from several cards, this mode is hardly suitable, since for each card from the group it requires at least 2 additional cores (with `dpdk_rss`=2). It is better to consider the options `dpdk_engine=1` or `dpdk_engine=3`.

Stingray SG configures cards as follows:

- RX queue count = `dpdk_rss`
- TX queue count = number of processing threads. Processing threads record data directly each to their TX-card queue.

dpdk_engine=3: Dispatcher for a bridge

A separate dispatcher thread is created for each bridge. Designed for configurations with multiple input and output devices:

```
in_dev=01-00.0:02-00.0:03-00.0
out_dev=01-00.1:02-00.1:03-00.1
dpdk_engine=3
```

In this example, three dispatcher threads are created:

- for bridge 01-00.0 ←→ 01-00.1

- for bridge 02-00.0 ←→ 02-00.1
- for bridge 03-00.0 ←→ 03-00.1

The total number of cores required is equal to the number of handlers plus the number of bridges.

This engine is designed for several 25G/40G/50G cards in a group (that is, for SSG-100+).

Stingray SG configures cards as follows:

- RX queue count = 1
- TX queue count = number of processing threads. Processing threads record data directly each to their TX-card queue.

dpdk_engine=4: Dispatcher for a port

A separate dispatcher thread is created for each port (device). Designed for configurations with a range of devices on input and output:

```
in_dev=01-00.0:02-00.0:03-00.0
out_dev=01-00.1:02-00.1:03-00.1
dpdk_engine=4
```

For this example, six dispatcher threads are created – one dispatcher for each device. Obviously, if we have only one bridge, this engine is equivalent to `dpdk_engine=1` – one dispatcher per direction.

The total number of cores required is equal to the number of handlers plus the number of ports

This engine is designed for multiple 25G/40G/50G cards in a group (i.e. for SSG-100+)

The SSG configures the cards as follows:

- RX queue count = 1
- TX queue count = The processing threads write directly each to its own TX queue card.