Содержание

3 DPDK Version 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 first step to work with DPDK is to take the network cards out of the control of the operating system. DPDK works with PCI devices, that can be displayed with the command:

```
> lspci|grep Eth
41:00.0 Ethernet controller: Intel Corporation Ethernet Controller XXV710
for 25GbE SFP28 (rev 02)
41:00.1 Ethernet controller: Intel Corporation Ethernet Controller XXV710
for 25GbE SFP28 (rev 02)
c6:00.0 Ethernet controller: Broadcom Inc. and subsidiaries BCM57416
NetXtreme-E Dual-Media 10G RDMA Ethernet Controller (rev 01)
c6:00.1 Ethernet controller: Broadcom Inc. and subsidiaries BCM57416
NetXtreme-E Dual-Media 10G RDMA Ethernet Controller (rev 01)
>
```

This command will list all PCI ethernet devices. Each line starts with the system PCI device identifier - these PCI identifiers are the unique for the network card in the DPDK.

Transferring the card to DPDK mode (disconnecting from the system network driver) is carried out by the dpdk-devbind.py utility from the DPDK:

```
# Example - devices 41:00.0 and 41:00.1 transfer to the DPDK mode

>insmod $RTE/module/igb_uio.ko

# 25G NICs
>$RTE/bin/dpdk-devbind.py --bind igb_uio 0000:41:00.0
>$RTE/bin/dpdk-devbind.py --bind igb_uio 0000:41:00.1
```

here, igb_uio - is UIO driver. The system `uio_pci_generic` or `igb_uio` from the DPDK can act as a uio driver. Usually `uio_pci_generic` is used for modern cards, and `igb_uio` for the older ones, for details see DPDK Linux Drivers. Uio-driver is only needed to register interrupts of network cards (e.g. to recognize link down/link up), and is not involved in receiving and sending data packets.

> When switching cards to DPDK mode, be careful not to accidentally switch the server's control interface to DPDK mode - the connection with the server will be interrupted immediately!

To see if the card is properly initialized to work with DPDK, use the command

```
> $RTE/bin/dpdk-devbind.py --status
```

If the cards are in DPDK mode, you will see them in `Network devices using DPDK-compatible driver` section:

```
> $RTE/bin/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
0000:41:00.0 'Ethernet Controller XXV710 for 25GbE SFP28 158b' drv=igb_uio
unused=i40e
0000:41:00.1 'Ethernet Controller XXV710 for 25GbE SFP28 158b' drv=igb_uio
unused=i40e
....
```

Also you have to reserve huge page:

```
#!/bin/bash

# Reserve 4 1G-pages - 4 GB in total:
HUGEPAGES_NUM=4
HUGEPAGES_PATH=/dev/hugepages
sync && echo 3 > /proc/sys/vm/drop_caches
echo $HUGEPAGES_NUM >
/sys/kernel/mm/hugepages/hugepages-1048576kB/nr_hugepages
HUGEPAGES_AVAIL=$(grep HugePages_Total
/sys/devices/system/node/node0/meminfo | cut -d ':' -f 2|sed 's/ //g')
if [ $HUGEPAGES_AVAIL -ne $HUGEPAGES_NUM ]; then
        printf "Warning: %s hugepages available, %s requested\n"
"$HUGEPAGES_AVAIL" "$HUGEPAGES_NUM"
fi
```

Usually 2-4 GB for a huge page is enough for the normal functioning of Stingray SG. If it is not enough, Stingray SG will display a critical error in fastdpi_alert.log and will not start. All the memory necessary for the operation of Stingray SG is allocated when starting from the huge page, so if the SSG has started with the current settings, the system will not need more and more memory from the huge page. In case of startup errors associated with a shortage of huge pages, you need to increase the number of allocated huge pages in the script above and try to run the Stingray SG again.

All these actions - transferring cards into DPDK mode and reserving the huge page - must be performed at OS startup.

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.

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`).