NVIDIA DOCA Switch Application Guide
This guide provides an example of switch implementation on top of NVIDIA® BlueField® DPU.

**Introduction**

DOCA Switch is a network application that leverages the DPU’s hardware capability for internal switching between representor ports on the DPU.

DOCA Switch is based on the DOCA Flow library. As such, it exposes a command line interface which receives DOCA Flow like commands to allow adding rules in real time.

**System Design**

DOCA Switch is designed to run on the DPU as a standalone application (all network traffic goes directly through it).

Traffic flows between two VMs on the host:
Traffic flow from a physical port to a VM on the host:
Application Architecture

DOCA Switch is based on 3 modules:

- Command line interface – receives pre-defined DOCA Flow-like commands and parses them
- Flow pipes manager – generates a unique identification number for each DOCA Flow structure created
- Switch core – combines all modules together and calls necessary DOCA Flow API
Port initialization cannot be made dynamically. All ports must be defined when running the application with standard DPDK flags.

- When adding a pipe or an entry, the user must run commands to create the relevant structs beforehand

- Optional parameters must be specified by the user in the command line; otherwise, NULL is used

- After a pipe or an entry is created successfully, the relevant ID is printed for future use

**DOCA Libraries**
This application leverages the following DOCA libraries:

- DOCA Flow

Refer to its respective programming guide for more information.

**Compiling the Application**

ℹ️ **Info**

Please refer to the NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

The installation of DOCA’s reference applications contains the sources of the applications, alongside the matching compilation instructions. This allows for compiling the applications "as-is" and provides the ability to modify the sources, then compile a new version of the application.

💡 **Tip**

For more information about the applications as well as development and compilation tips, refer to the DOCA Applications page.

The sources of the application can be found under the application's directory: `:/opt/mellanox/doca/applications/switch/`.

**Compiling All Applications**

All DOCA applications are defined under a single meson project. So, by default, the compilation includes all of them.

To build all the applications together, run:
Recompiling Only the Current Application

To directly build only the switch application:

```
cd /opt/mellanox/doca/applications/
meson /tmp/build -Denable_allApplications=false -Denable_switch=true
ninja -C /tmp/build
```

Info

doca_switch is created under /tmp/build/switch/.

Alternatively, one can set the desired flags in the meson_options.txt file instead of providing them in the compilation command line:

1. Edit the following flags in /opt/mellanox/doca/applications/meson_options.txt:
   - Set enable_allApplications to false
Set `enable_switch` to `true`.

2. Run the following compilation commands:

```
   cd /opt/mellanox/doca/applications/
   meson /tmp/build
   ninja -C /tmp/build
```

**Info**

doca_switch is created under `/tmp/build/switch/`.

**Troubleshooting**

Refer to the [NVIDIA DOCA Troubleshooting Guide](#) for any issue encountered with the compilation of the application.

**Running the Application**

**Prerequisites**

The switch application is based on DOCA Flow. Therefore, the user is required to allocate huge pages.

```
   echo '2048' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

**Note**
On some operating systems (RockyLinux, OpenEuler, CentOS 8.2) the default huge page size on the DPU (and Arm hosts) is larger than 2MB, and is often 512MB instead. Once can find out the size of the huge pages using the following command:

```
$ grep -i huge /proc/meminfo
```

```
AnonHugePages: 0 kB
ShmemHugePages: 0 kB
FileHugePages: 0 kB
HugePages_Total: 4
HugePages_Free: 4
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 524288 kB
Hugetlb: 6291456 kB
```

Given that the guiding principal is to allocate 4GB of RAM, in such cases instead of allocating 2048 pages, one should allocate the matching amount (8 pages):

```
echo '8' | sudo tee -a /sys/kernel/mm/hugepages/hugepages-
524288kB/nr_hugepages
```

**Application Execution**

The switch application is provided in source form. Therefore, hence a compilation is required before the application can be executed.

1. Application usage instructions:

   Usage: doca_switch [DPDK Flags] -- [DOCA Flags]
2. CLI example for running the application on the BlueField:

```
./doca_switch -a 03:00.0,representor=[0-2],dv_flow_en=2 -- -l 60
```
**Note**

The PCIe address (03:00.0) should match the address of the desired PCIe device.

---

### Command Line Flags

<table>
<thead>
<tr>
<th>Flag Type</th>
<th>Short Flag</th>
<th>Long Flag</th>
<th>Description</th>
<th>JSON Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>General flags</td>
<td>h</td>
<td>help</td>
<td>Prints a help synopsis</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>v</td>
<td>version</td>
<td>Prints program version information</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>l</td>
<td>log-level</td>
<td>Set the log level for the application:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DISABLE=10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CRITICAL=20</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ERROR=30</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• WARNING=40</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• INFO=50</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DEBUG=60</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• TRACE=70 (requires compilation with TRACE log level support)</td>
<td>&quot;log-level&quot;: 60</td>
</tr>
<tr>
<td>N/A</td>
<td>sdk-log-level</td>
<td>Sets the log level for the program:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DISABLE=10</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CRITICAL=20</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• ERROR=30</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• WARNING=40</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• INFO=50</td>
<td>&quot;sdk-log-level&quot;: 40</td>
</tr>
<tr>
<td>Flag Type</td>
<td>Short Flag</td>
<td>Long Flag</td>
<td>Description</td>
<td>JSON Content</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>-----------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>j</td>
<td>JSON Content</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### Info

Refer to DOCA Arg Parser for more information regarding the supported flags and execution modes.

### Supported Commands

- create pipe port_id=[port_id][,<optional_parameters>]
  - Available optional parameters:
    - name=<pipe-name>
    - root_enable=[1|0]
    - monitor=[1|0]
    - match_mask=[1|0]
    - fwd=[1|0]
    - fwd_miss=[1|0]
    - type=[basic|control]

- add entry pipe_id=<pipe_id>,pipe_queue=<pipe_queue>[,<optional_parameters>]
  - Available optional parameters:
    - monitor=[1|0]
- `fwd=[1 | 0]`

- `add control_pipe entry priority=<priority>,pipe_id=<pipe_id>,pipe_queue=<pipe_queue>[, <optional_parameters>]`

**Available optional parameters:**

- `match_mask=[1 | 0]`

- `fwd=[1 | 0]`

- `destroy pipe pipe_id=<pipe_id>`

- `rm entry pipe_queue=<pipe_queue>,entry_id=[entry_id]`

- `port pipes flush port_id=[port_id]`

- `port pipes dump port_id=[port_id],file=[file_name]`

- `query entry_id=[entry_id]`

- `create [struct] [field=value,...]`

**Struct options:** `pipe_match`, `entry_match`, `match_mask`, `actions`, `monitor`, `fwd`, `fwd_miss`

- **Match struct fields:**

<table>
<thead>
<tr>
<th>Fields</th>
<th>Field Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>port_meta</td>
<td></td>
</tr>
<tr>
<td>outer.eth.src_mac</td>
<td></td>
</tr>
<tr>
<td>outer.eth.dst_mac</td>
<td></td>
</tr>
<tr>
<td>outer.eth.type</td>
<td></td>
</tr>
<tr>
<td>outer.vlan_tci</td>
<td></td>
</tr>
<tr>
<td>outer.l3_type</td>
<td><code>ipv4, ipv6</code></td>
</tr>
<tr>
<td>outer.src_ip_addr</td>
<td></td>
</tr>
<tr>
<td>outer.dst_ip_addr</td>
<td></td>
</tr>
<tr>
<td>outer.l4_type_ext</td>
<td><code>tcp, udp, gre</code></td>
</tr>
<tr>
<td>outer.tcp.flags</td>
<td><code>FIN, SYN, RST, PSH, ACK, URG, ECE, CWR</code></td>
</tr>
</tbody>
</table>
### Fields

<table>
<thead>
<tr>
<th>Fields</th>
<th>Field Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>outer.tcp_src_port</td>
<td></td>
</tr>
<tr>
<td>outer.tcp_dst_port</td>
<td></td>
</tr>
<tr>
<td>outer.udp_src_port</td>
<td></td>
</tr>
<tr>
<td>outer.udp_dst_port</td>
<td></td>
</tr>
<tr>
<td>tun_type</td>
<td></td>
</tr>
<tr>
<td>vxlan_tun_id</td>
<td></td>
</tr>
<tr>
<td>gre_key</td>
<td></td>
</tr>
<tr>
<td>gtp_teid</td>
<td></td>
</tr>
<tr>
<td>inner.eth.src_mac</td>
<td></td>
</tr>
<tr>
<td>inner.eth.dst_mac</td>
<td></td>
</tr>
<tr>
<td>inner.eth.type</td>
<td></td>
</tr>
<tr>
<td>inner.vlan_tci</td>
<td></td>
</tr>
<tr>
<td>inner.l3_type</td>
<td>ipv4, ipv6</td>
</tr>
<tr>
<td>inner.src_ip_addr</td>
<td></td>
</tr>
<tr>
<td>inner.dst_ip_addr</td>
<td></td>
</tr>
<tr>
<td>inner.l4_type_ext</td>
<td>tcp, udp</td>
</tr>
<tr>
<td>inner.tcp.flags</td>
<td>FIN, SYN, RST, PSH, ACK, URG, ECE, CWR</td>
</tr>
<tr>
<td>inner.tcp_src_port</td>
<td></td>
</tr>
<tr>
<td>inner.tcp_dst_port</td>
<td></td>
</tr>
<tr>
<td>inner.udp_src_port</td>
<td></td>
</tr>
<tr>
<td>inner.udp_dst_port</td>
<td></td>
</tr>
</tbody>
</table>

### Actions struct fields:

<table>
<thead>
<tr>
<th>Fields</th>
<th>Field Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>decap</td>
<td>true, false</td>
</tr>
<tr>
<td>mod_src_mac</td>
<td></td>
</tr>
<tr>
<td>mod_dst_mac</td>
<td></td>
</tr>
<tr>
<td>mod_src_ip_type</td>
<td>ipv4, ipv6</td>
</tr>
<tr>
<td>mod_src_ip_addr</td>
<td></td>
</tr>
</tbody>
</table>
Fields | Field Options
---|---
mod_dst_ip_type | ipv4, ipv6
mod_dst_ip_addr | 
mod_src_port | 
mod_dst_port | 
ttl | 
has_encap | true, false
encap_src_mac | 
encap_dst_mac | 
encap_src_ip_type | ipv4, ipv6
encap_src_ip_addr | 
encap_dst_ip_type | ipv4, ipv6
encap_dst_ip_addr | 
encap_tup_type | vxlan, gtpu, gre
encap_vxlan-tun_id | 
encap_gre_key | 
encap_gtp_teid | 

- **FWD struct fields:**

Fields | Field Options
---|---
type | rss, port, pipe, drop
rss_flags | 
rss_queues | 
num_of_queues | 
port_id | 
next_pipe_id | 

- **Monitor struct fields:**
  - flags
The physical port number (only one physical port is supported) will always be 0 and all representor ports are numbered from 1 to N where N is the number of representors being used. For example:

- Physical port ID: 0
- VF0 representor port ID: 1
- VF1 representor port ID: 2
- VF2 representor port ID: 3

The following is an example of creating a pipe and adding one entry into it:

```
create fwd type=port,port_id=0xffff
create pipe port_id=0,name=p0_to_vf1,root_enable=1,fwd=1
create fwd type=port,port_id=1
add entry pipe_queue=0,fwd=1,pipe_id=1012
....
rm entry pipe_queue=0,entry_id=447
```

1. Pipe is configured on port ID 0 (physical port).
2. Entry is configured to forward all traffic directly into port ID 1 (VF0).
3. When the forwarding rule is no longer needed, the entry is deleted.
4. Ultimately, both entries are deleted, each according to the unique random ID it was given:

**Troubleshooting**
Refer to the NVIDIA DOCA Troubleshooting Guide for any issue encountered with the installation or execution of the DOCA applications.

**Application Code Flow**

1. Parse application argument.
   
   1. Initialize the arg parser resources and register DOCA general parameters.

   ```
   doca_argp_init();
   ```

   2. Register application parameters.

   ```
   register_switch_params();
   ```

   3. Parse app parameters.

   ```
   doca_argp_start();
   ```

2. Count total number of ports.

   ```
   switch_ports_count();
   ```

   1. Check how many ports are entered when running the application.

3. Initialize DPDK ports and queues.

   ```
   dpdk_queues_and_ports_init();
   ```
4. Initialize DOCA Switch.

```c
switch_init();
```

1. Initialize DOCA Flow.

2. Create port pairs.


4. Register an action for each relevant CLI command.

5. Initialize Flow Parser.

```c
flow_parser_init();
```

1. Reset all internal Flow Parser structures.

2. Start the command line interface.

3. Receive user commands, parse them, and call the required DOCA Flow API command.

4. Close the interactive shell once a "quit" command is entered.

6. Clean Flow Parser resources.

```c
flow_parser_cleanup();
```

7. Destroy Switch resources.

```c
switch_destroy();
```

1. Destroy Flow Pipes Manager resources.

```c
switch_destroy();
```


```c
dpk queues_and_ports_fini();
```

10. DPDK finish.

```c
dpk fini();
```

1. Call `rte_eal_destroy()` to destroy initialized EAL resources.

11. Arg parser destroy.

```c
doca_argp_destroy();
```

**References**

- /opt/mellanox/doca/applications/switch/

© Copyright 2024, NVIDIA. PDF Generated on 08/15/2024