Table of contents

Introduction

Prerequisites

API

enum doca_flow_tune_server_kpi_type

struct doca_flow_tune_server_shared_resources_kpi_res

struct doca_flow_tune_server_kpi_res

doca_flow_tune_server_cfg_create

doca_flow_tune_server_cfg_set_bind_path

doca_flow_tune_server_cfg_destroy

doca_flow_tune_server_init

doca_flow_tune_server_destroy

doca_flow_tune_server_query_pipe_line

doca_flow_tune_server_get_port_ids

doca_flow_tune_server_get_kpi

doca_flow_tune_server_get_port_kpi

DOCA Flow Tune Server Samples

Running the Samples

Samples

Flow Tune Server KPI

Flow Tune Server Dump Pipeline

Flow Visualization
This guide provides an overview and configuration instructions for DOCA Flow Tune Server API.

**Introduction**

DOCA Flow Tune Server (TS), DOCA Flow subcomponent, exposes an API to collect predefined internal key performance indicators (KPIs) and pipeline visualization of a running DOCA Flow application.

Supported port KPIs:

- Total add operations across all queues
- Total update operations across all queues
- Total remove operations across all queues
- Pending operations number across all queues
- Number of *NO_WAIT* flag operations across all queues
- Number of shared resources and counters
- Number of pipes

Supported application KPIs:

- Number of ports
- Number of queues
- Queues depth

Pipeline information is saved to a JSON file to simplify its structure. Visualization is supported for the following DOCA Flow pipes:

- Basic
- Control

Each pipe contains the following fields:
- Type
- Name
- Domain
- Is root
- Match
- Match mask
- FWD
- FWD miss

Supported entry information:

- Basic
  - FWD
- Control
  - FWD
  - Match
  - Match mask
  - Priority

**Prerequisites**

DOCA Flow Tune Server API is available only by using the DOCA Flow and DOCA Flow Tune Server trace libraries.
The following subsections provide additional details about the library API.

**enum doca_flow_tune_server_kpi_type**

DOCA Flow TS KPI flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NR_PORTS,</td>
<td>Retrieve port number</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NR_QUEUES,</td>
<td>Retrieve queue number</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_QUEUE_DEPTH,</td>
<td>Retrieve queue depth</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NR_SHARED_RESOURCES,</td>
<td>Retrieve shared resource and counter numbers</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NRPIPES,</td>
<td>Retrieve number of pipes per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_ADD,</td>
<td>Retrieve entry add operations per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_UPDATE,</td>
<td>Retrieve entry update operations per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_ENTRIES_OPS_REMOVE,</td>
<td>Retrieve entry remove operations per port</td>
</tr>
<tr>
<td>TUNE_SERVER_KPI_TYPE_PENDING_OPS,</td>
<td>Retrieve entry pending operations per port</td>
</tr>
</tbody>
</table>

For more detailed information, refer to section "DOCA Flow Debug and Trace" under DOCA Flow.
<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TUNE_SERVER_KPI_TYPE_NO_WAIT_OPS,</td>
<td>Retrieve entry NO_WAIT flag operations per port</td>
</tr>
</tbody>
</table>

### struct doca_flow_tune_server_shared_resources_kpi_res

Holds the number of each shared resources and counters per port.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint64_t nr_meter</td>
<td>Number of meters</td>
</tr>
<tr>
<td>uint64_t nr_counter</td>
<td>Number of counters</td>
</tr>
<tr>
<td>uint64_t nr_rss</td>
<td>Number of RSS</td>
</tr>
<tr>
<td>uint64_t nr_mirror</td>
<td>Number of mirrors</td>
</tr>
<tr>
<td>uint64_t nr_psp</td>
<td>Number of PSP</td>
</tr>
<tr>
<td>uint64_t nr_encap</td>
<td>Number of encap</td>
</tr>
<tr>
<td>uint64_t nr_decap</td>
<td>Number of decap</td>
</tr>
</tbody>
</table>

### struct doca_flow_tune_server_kpi_res

Holds the KPI result.

#### Note

This structure is required when calling doca_flow_tune_server_get_kpi or doca_flow_tune_server_get_port_kpi.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enum doca_flow_tune_server_kpi_type type</td>
<td>KPI result type</td>
</tr>
<tr>
<td>struct doca_flow_tune_server_shared_resources_kpi_res</td>
<td>Shared resource result</td>
</tr>
<tr>
<td>shared_resources_kpi</td>
<td>values</td>
</tr>
<tr>
<td>uint64_t val</td>
<td>Result value</td>
</tr>
</tbody>
</table>

**doca_flow_tune_server_cfg_create**

Creates DOCA Flow Tune Server configuration structure.

```c
doca_error_t doca_flow_tune_server_cfg_create(struct doca_flow_tune_server **cfg);
```

**doca_flow_tune_server_cfg_set_bind_path**

Adds local path to the configuration struct on which the DOCA Flow Tune Server AF_UNIX socket binds.

```c
doca_error_t doca_flow_tune_server_cfg_set_bind_path(struct doca_flow_tune_server *cfg, const char *path, size_t path_len);
```

**doca_flow_tune_server_cfg_destroy**

Destroys DOCA Flow Tune Server configuration structure.

```c
doca_error_t doca_flow_tune_server_cfg_destroy(struct doca_flow_tune_server *cfg);
```
**doca_flow_tune_server_init**

Initializes DOCA Flow Tune Server internal structures.

```
doca_error_t doca_flow_tune_server_init(void);
```

**doca_flow_tune_server_destroy**

Destroys DOCA Flow Tune Server internal structures.

```
void doca_flow_tune_server_destroy(void);
```

**doca_flow_tune_server_query_pipe_line**

Queries and dumps pipeline info for all ports to a JSON file pointed by fp.

```
doca_error_t doca_flow_tune_server_query_pipe_line(FILE *fp);
```

**doca_flow_tune_server_get_port_ids**

Retrieves ports identification numbers.

```
doca_error_t doca_flow_tune_server_get_port_ids(uint16_t *port_id_arr, uint16_t port_id_arr_len, uint16_t *nr_ports);
```

**doca_flow_tune_server_get_kpi**
Retrieves application scope KPI.

doca_error_t doca_flow_tune_server_get_kpi(enum doca_flow_tune_server_kpi_type kpi_type,
                                          struct doca_flow_tune_server_kpi_res *res)

doca_flow_tune_server_get_port_kpi

Retrieves port scope KPI.

doca_error_t doca_flow_tune_server_get_port_kpi(uint16_t port_id,
                                                  enum doca_flow_tune_server_kpi_type kpi_type,
                                                  struct doca_flow_tune_server_kpi_res *res);

DOCA Flow Tune Server Samples

This section describes DOCA Flow Tune Server samples.

The samples illustrate how to use the library API to retrieve KPIs or save pipeline information into a JSON file.

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Running the Samples

1. Refer to the following documents:
NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.

NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA samples.

2. To build a given sample:

```bash
cd /opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline
meson /tmp/build
ninja -C /tmp/build
```

**Info**

The binary `doca_flow_tune_server_dump_pipeline` is created under `/tmp/build/samples/`.

3. Sample (e.g., `doca_flow_tune_server_dump_pipeline`) usage:

Usage: `doca_<sample_name> [DOCA Flags] [Program Flags]

DOCA Flags:
- `-h, --help` Print a help synopsis
- `-v, --version` Print program version information
- `-l, --log-level` Set the (numeric) log level for the program: 10=DISABLE, 20=Critical, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE
- `--sdk-log-level` Set the SDK (numeric) log level for the program: 10=DISABLE, 20=Critical, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE
- `-j, --json <path>` Parse all command flags from an input json file

4. For additional information per sample, use the `-h` option:
5. The following is a CLI example for running the samples:

```
/tmp/build/doca_<sample_name> -a auxiliary:mlx5_core.sf.2,dv_flow_en=2 -a auxiliary:mlx5_core.sf.3,dv_flow_en=2 -- -l 60
```

### Samples

#### Flow Tune Server KPI

This sample illustrates how to use DOCA Flow Tune Server API to retrieve KPIs.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.

2. Starting a single DOCA Flow port.

3. Creating a server configuration struct using the `doca_flow_tune_server_cfg_create` function.

4. Initializing DOCA Flow server using the `doca_flow_tune_server_init` function. This must be done after calling the `doca_flow_port_start` function (or the `init_doca_flow_ports` helper function).

5. Querying existing port IDs using the `doca_flow_tune_server_get_port_ids` function.

6. Querying application level KPIs using `doca_flow_tune_server_get_kpi` function. The following KPI are read:
   - Number of queues
   - Queue depth

7. KPIs per port on which the basic pipe is created:
1. Add operation entries.

8. Adding 20 entries followed by a second call to query entries add operations.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/flow_tune_server_kpi_sample.c
- /opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/flow_tune_server_kpi_main.c
- /opt/mellanox/doca/samples/doca_flow/flow_tune_server_kpi/meson.build

### Flow Tune Server Dump Pipeline

This sample illustrates how to use DOCA Flow Tune Server API to dump pipeline information into a JSON file.

The sample logic includes:

1. Initializing DOCA Flow by indicating `mode_args="vnf,hws"` in the `doca_flow_cfg` struct.
2. Starting two DOCA Flow ports.
3. Creating server configuration struct using the `doca_flow_tune_server_cfg_create` function.

**Note**

This must be done after calling `init_foca_flow_ports` function.

5. Opening a file called `sample_pipeline.json` for writing.
6. For each port:
   1. Creating a pipe to drop all traffic.
2. Creating a pipe to hairpin traffic from port 0 to port 1

3. Creating FWD pipe to forward traffic based on 5-tuple.

4. Adding two entries to FWD pipe, each entry with different 5-tuple.

5. Creating a control pipe and adding the FWD pipe as an entry.

7. Dumping the pipeline information into a file.

Reference:

- /opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/flow_tune_server_dump_pipeline
- /opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/flow_tune_server_dump_pipeline
- /opt/mellanox/doca/samples/doca_flow/flow_tune_server_dump_pipeline/meson.build

**Flow Visualization**

Once a DOCA Flow application pipeline has been exported to a JSON file, it is easy to visualize it using tools such as Mermaid.

1. Save the following Python script locally to a file named `doca-flow-viz.py` (or similar). This script converts a given JSON file produced by DOCA Flow TS to a Mermaid diagram embedded in a markdown document.

```python
#!/usr/bin/python3

#
# Copyright (c) 2024 NVIDIA CORPORATION & AFFILIATES, ALL RIGHTS RESERVED.
#
# This software product is a proprietary product of NVIDIA CORPORATION &
# AFFILIATES (the "Company") and all right, title, and interest in and to the
# software product, including all associated intellectual property rights, are
# and shall remain exclusively with the Company.
#
# This software product is governed by the End User License Agreement
```
# provided with the software product.
#

```python
import glob
import json
import sys
import os.path

class MermaidConfig:
    def __init__(self):
        self.prefix_pipe_name_with_port_id = False
        self.show_match_criteria = False
        self.show_actions = False

class MermaidFormatter:
    def __init__(self, cfg):
        self.cfg = cfg
        self.syntax = ""n
        self.prefix_pipe_name_with_port_id = cfg.prefix_pipe_name_with_port_id

    def format(self, data):
        self.prefix_pipe_name_with_port_id = self.cfg.prefix_pipe_name_with_port_id
        and len(data.get('ports', [])) > 0

        if not 'ports' in data:
            port_id = data.get('port_id', 0)
            data = {
                'ports': [
                    {n 'port_id': port_id,
                    'pipes': data['pipes']}
                ]
            }

        self.syntax = ""
        self.append("```mermaid")
        self.append('graph LR')
        self.append('declare终端状态(data)

        for port in data['ports']:
            self.process_port(port)

        self.append("```")
```
return self.syntax
def append(self, text, endline = "\n"):
    self.syntax += text + endline
def declare_terminal_states(self, data):
    all_fwd_types = self.get_all_fwd_types(data)
    if 'drop' in all_fwd_types:
        self.append('    drop[[drop]]')
    if 'rss' in all_fwd_types:
        self.append('    RSS[[RSS]]')

def get_all_fwd_types(self, data):
    # Gather all 'fwd' and 'fwd_miss' types from pipes and 'fwd' types from entries
    all_fwd_types = {
        fwd_type
        for port in data.get('ports', [])
        for pipe in port.get('pipes', [])
        for tag in ['fwd', 'fwd_miss'] # Process both 'fwd' and 'fwd_miss' for each pipe
        for fwd_type in [pipe.get(tag, {}).get('type', None)] # Extract the 'type'
        if fwd_type
    } | {
        fwd_type
        for port in data.get('ports', [])
        for pipe in port.get('pipes', [])
        for tag in ['fwd']
        for entry in pipe.get('entries', []) # Process all entries in each pipe
        for fwd_type in [entry.get(tag, {}).get('type', None)]
        if fwd_type
    }
    return all_fwd_types

def process_port(self, port):
    port_id = port['port_id']
    pipe_names = self.resolve_pipe_names(port)
    self.declare_pipes(port, pipe_names)
    for pipe in port.get('pipes', []):
        self.process_pipe(pipe, port_id)

def resolve_pipe_names(self, port):
    pipe_names = {}
    port_id = port['port_id']
    for pipe in port.get('pipes', []):
        id = pipe['pipe_id']
        name = pipe['attributes'].get('name', f"pipe_{id}")
        if self.prefix_pipe_name_with_port_id:
name = f"p(port_id).{name}"
pipe_names[id] = name
return pipe_names
def declare_pipes(self, port, pipe_names):
    port_id = port['port_id']
    for pipe in port.get('pipes', []):
        id = pipe['pipe_id']
        name = pipe_names[id]
        self.declare_pipe(port_id, pipe, name)
def declare_pipe(self, port_id, pipe, pipe_name):
    id = pipe['pipe_id']
    attr = "\n(root)" if self.pipe_is_root(pipe) else ""
    if self.cfg.show_match_criteria and not self.pipe_is_ctrl(pipe):
        fields_matched = self.pipe_match_criteria(pipe, 'match')
        attr += f"\nmatch: {fields_matched}"
        self.append(f"p{port_id}.pipe_{id}{{{pipe_name}{attr}}}"
    
def pipe_match_criteria(self, pipe, key: [‘match’, ‘match_mask’]):
        return 
    
def extract_match_criteria_paths(self, prefix, match):
        for k,v in match.items:
            if isinstance(v, dict):
                new_prefix = f"{prefix}.{k}" if prefix else k
                for x in self.extract_match_criteria_paths(new_prefix, v):
                    yield x
            else:
                # ignore v, the match value
                yield f"{prefix}.{k}" if prefix else k

def pipe_is_ctrl(self, pipe):
    return pipe['attributes']['type'] == 'control'

def pipe_is_root(self, pipe):
    return pipe['attributes'].get('is_root', False)

def process_pipe(self, pipe, port_id):
    pipe_id = f"pipe_{pipe['pipe_id']}"
    is_ctrl = self.pipe_is_ctrl(pipe)
    self.declare_fwd(port_id, pipe_id, '-->', self.get_fwd_target(pipe.get('fwd', {}), port_id))
    self.declare_fwd(port_id, pipe_id, '-.->', self.get_fwd_target(pipe.get('fwd_miss', {}), port_id))
    for entry in pipe.get('entries', []):
        fields_matched = self.pipe_match_criteria(entry, 'match') if is_ctrl else None
        fields_matched = f"{|fields_matched|}" if fields_matched else ""
self.declare_fwd(port_id, pipe_id, f'-->{fields_matched}'), self.get_fwd_target(entry.get('fwd', {}), port_id))

    if self.pipe_is_root(pipe):
        self.declare_fwd(port_id, None, '->', f'p{port_id}.pipe_{pipe_id}'

def get_fwd_target(self, fwd, port_id):
    fwd_type = fwd.get('type', None)
    if not fwd_type:
        return None
    elif fwd_type == 'changeable':
        return None
    elif fwd_type == 'pipe':
        pipe_id = fwd.get('pipe_id', fwd.get('value', None))
        target = f'p{port_id}.pipe_{pipe_id}'
    elif fwd_type == 'port':
        port_id = fwd.get('port_id', fwd.get('value', None))
        target = f'p{port_id}.egress'
    else:
        target = f'{fwd_type}'
    return target

def declare_fwd(self, port_id, pipe_id, arrow, target):
    if target:
        src = f'p{port_id}.pipe_{pipe_id}' if pipe_id else f'p{port_id}.ingress'
        self.append(f'    {src} {arrow} {target}')

def json_to_md(infile, outfile, cfg):
    formatter = MermaidFormatter(cfg)
    data = json.load(infile)
    mermaid_syntax = formatter.format(data)
    outfile.write(mermaid_syntax)

def json_dir_to_md_inplace(dir, cfg):
    for infile in glob.glob(dir + '/**/*.json', recursive=True):
        outfile = os.path.splitext(infile)[0] + '.md'
        print(f'"{infile} --> {outfile}"')
        json_to_md(open(infile, 'r'), open(outfile, 'w'), cfg)

def main()->int:
    cfg = MermaidConfig()
    cfg.show_match_criteria = True

    if len(sys.argv) == 2 and os.path.isdir(sys.argv[1]):
        json_dir_to_md_inplace(sys.argv[1], cfg)
2. The resulting Markdown can be viewed in several ways, including:

- Microsoft Visual Studio Code (using an available Mermaid plugin, such as this [one](#))
- In the GitHub and GitLab built-in Markdown renderer (after committing the output to a Git repo)
- By pasting only the Flowchart content into the [Online FlowChart and Diagram Editor](#)

3. The Python script can be invoked as follows:

```python
python3 doca-flow-viz.py sample_pipeline.json sample_pipeline.md
```

In the case of the `flow_tune_server_dump_pipeline` sample, the script produces the following diagram: