NAME

ovs-flowviz – utility for visualizing OpenFlow and datapath flows

SYNOPSIS

ovs−flowviz [**−i** [*alias*,]*file* | **−−input** [*alias*,]*file*] [**−c** *file* | **−−config** *file*] [**−f** *filter* | **−−filter** *filter*] [**−l** *filter* | **−−highlight** *filter*] [**−−style** *style*] *flow−type format* [*args*…]

ovs−flowviz −−help

DESCRIPTION

ovs−flowviz helps visualize OpenFlow and datapath flow dumps in different formats in order to make them more easily understood.

ovs−flowviz reads flows from **stdin** or from a *file* specified by the **−−input** option, filters them, highlights them, and finally outputs them in one of the predefined *format*s.

OPTIONS

−h, −−help
: Print a brief help message to the console.

−i [<alias>,]<file>, −−input [<alias>,]<file>
: File to read flows from. If not provided, **ovs−flowviz** will read flows from stdin.

  This option can be specified multiple times. The file path can prepended by an alias that will be shown in the output. For example: **−−input node1,/path/to/file1 −−input node2,/path/to/file2**

−c <file>, −−config <file>
: Style configuration file to use, overriding the default one. Styles defined in the style configuration file can be selected using the **−−style** option.

  For more details on the style configuration file, see the *Style Configuration File* section below.

−f <filter>, −−filter <filter>
: Flow filter expression. Only those flows matching the expression will be shown (although some formats implement filtering differently, see the *Datapath tree format* section below).

  The filtering syntax is detailed in *Filtering Syntax*.

−l <filter>, −−highlight <filter>
: Highlight the flows that match the provided *filter* expression.

  The filtering syntax is detailed in *Filtering Syntax*.

−−style <style>
: Style. The selected *style* must be defined in the style configuration file.

flow−type
: **openflow** or **datapath**.

format See the *Supported formats* section.

SUPPORTED FORMATS

ovs−flowviz supports several visualization formats for both OpenFlow and datapath flows:

Flow Type	Format	Description
Both	console	Prints the flows in a configurable, colorful style in the console.
Both	json	Prints the flows in JSON format.
Both	html	Prints the flows in an HTML list.
OpenFlow	cookie	Prints the flows in the console sorted by cookie.
OpenFlow	logic	Prints the logical structure of flows in the console.
Datapath	tree	Prints the flows as a tree structure arranged by **recirc_id** and **in_port**.
Datapath	graph	Prints a graphviz graph of the flows arranged by **recirc_id** and **in_port**.

Console format

The **console** format works for both OpenFlow and datapath flow types, and prints flows in the terminal using the style determined by the **−−style** option.

Arguments:

−h, −−heat−map

Color of the packet and byte counters to reflect their relative size. The color gradient goes through the following colors: blue (coldest, lowest), cyan, green, yellow, red (hottest, highest)

Note filtering is applied before the range is calculated.

JSON format

The **json** format works for both OpenFlow and datapath flow types, and prints flows in JSON format. See the *JSON Syntax* section for more details.

HTML format

The **html** format works for both OpenFlow and datapath flows, and prints flows in an HTML table that offers some basic interactivity. OpenFlow flows are sorted in tables and datapath flows are arranged in flow trees (see *Datapath tree format* for more details).

Styles defined via Style Configuration File and selected via **−−style** option also apply to the **html** format.

OpenFlow cookie format

The OpenFlow **cookie** format is similar to the **console** format but instead of arranging the flows by table, it arranges the flows by cookie.

OpenFlow logic format

The OpenFlow **logic** format helps visualize the logic structure of OpenFlow pipelines by arranging flows into *logical blocks*. A logical block is a set of flows that have:

- Same **priority**.
- Match on the same fields (regardless of the match value and mask).
- Execute the same actions (regardless of the actions' arguments, except for resubmit and output).
- Optionally, the **cookie** can be included as part of the logical flow.

Arguments:

−s, −−show−flows
Show all the flows under each logical block.

−d, −−ovn−detrace
Use ovn−detrace.py script to extract cookie information (implies '−c').

−c, −−cookie
Consider the cookie in the logical block.

−−ovn−detrace−path <path>
Use an alternative path to search for ovn_detrace.py.

−−ovnnb−db <conn>
OVN NB database connection method (implies '−d'). Default: "unix:/var/run/ovn/ovnnb_db.sock".

−−ovnsb−db <conn>
OVN SB database connection method (implies '−d'). Default: "unix:/var/run/ovn/ovnsb_db.sock".

−−o <filter>, −−ovn−filter <filter>
Specify the filter to be run on the ovn−detrace information. Syntax: python regular expression (See *https://docs.python.org/3/library/re.html*).

−h, −−heat−map
Change the color of the packet and byte counters to reflect their relative size. The color gradient goes through the following colors: blue (coldest, lowest), cyan, green, yellow, red (hottest, highest)

Note filtering is applied before the range is calculated.

Datapath tree format
The datapath **tree** format arranges datapath flows in a hierarchical tree. The tree is comprised of blocks with the same **recirc_id** and **in_port**. Within those blocks, flows with the same action are combined. And matches which are the same are omitted to reduce the visual noise.

When a flow's actions includes the **recirc()** action with a specific **recirc_id**, flows matching on that **recirc_id** and the same **in_port** are listed below. This is done recursively for all actions.

The result is a hierarchical representation that shows how actions are related to each other via recirculation. Note that flows with a specific non−zero **recirc_id** are listed below each group of flows that have a corresponding **recirc()** action. Therefore, the output contains duplicated flows and can be verbose.

Filtering works in a slightly different way for datapath flow trees. Unlike other formats where a filter simply removes non−matching flows, the output of a filtered datapath flow tree will show full sub−trees that contain at least one flow that satisfies the filter.

The **html** format prints this same tree as an interactive HTML table and the **graph** format shows the same tree as a graphviz graph.

Datapath graph format
The datapath **graph** generates a graphviz visual representation of the same tree−like flow hierarchy that the **tree** format prints.

Arguments:

−h, −−html
Print the graphviz format as an svg image alongside an interactive HTML table of flows.

JSON SYNTAX

Printing a single−file OpenFlow or datapath dump without PMD thread blocks in **json** format results in a list of JSON objects, each representing a flow.

This list can be found inside one or more levels of JSON dictionaries if multiple files are processed (filename used as key) or if PMD thread blocks are found in datapath flows (name of the thread used as key).

Each flow object includes the following keys:

orig Original flow string.

info Object with the flow information such as: cookie, duration, table, n_packets, n_bytes, etc.

match Object with the flow match. For each match, the object contains a key−value where the key is the name of the match as defined in **ovs−fields(7)** and **ovs−ofctl(8)**, and the value represents the match value. The way each value is represented depends on its type. See *Value representation*.

actions List of action objects. Each action is represented by an JSON object that has one key and one value. The key corresponds to the action name. The value represents the arguments of the key. See *Action representation*.

ufid The UFID (datapath flows only).

Value representation

Values are represented differently depending on their type:

- Flags: The value of flags is **true**.
- Decimal / Hexadecimal: Represented by their integer value. If they support masking, represented by a dictionary with two keys: **value** contains the field value and **mask** contains the mask. Both are integers.
- Ethernet: Represented by a string: **{address}[/{mask}]**
- IPv4 / IPv6: Represented by a string **{address}[/{mask}]**
- Registers: Represented by a dictionary with three keys: **field`** contains the field value (string), **start**, and **end** represent the first and last bit of the register value.

For example, the register

```
NXM_NX_REG10[0..15]
```

is represented as

```
{
    "field": "NXM_NX_REG10",
    "start": 0,
    "end": 15
},
```

Action representation

Actions are generally represented by an object that has a single key and value. The key is the action name as defined **ovs−actions(7)**.

The value of actions that have no arguments (such as **drop**) is (boolean) **true**.

The value of actions that have a list of arguments (e.g: **resubmit([port],[table],[ct])**) is an object that has the name of the argument as key. The argument names for each action is defined in ovs−actions. For example, the action

```
resubmit(,10)
```

is represented as

```
{
    "resubmit": {
```

```
            "port": "",
            "table": 10
        }
    }
```

The value of actions that have a key−word list as arguments (e.g: **ct([argument])**) is an object whose keys correspond to the keys defined in **ovs−actions(7)**. The way values are represented depends on the type of the argument. For example, the action

```
ct(table=14,zone=NXM_NX_REG12[0..15],nat)
```

is represented as

```
{
    "ct": {
        "table": 14,
        "zone": {
            "field": "NXM_NX_REG12",
            "start": 0,
            "end": 15
        },
        "nat": true
    }
}
```

STYLE CONFIGURATION FILE

The style configuration file is selected via the **−−config** option and has INI syntax. It can define any number of styles to be used by both **console** and **html** formats. Once defined in the configuration file, formats are selected using the **−−style** option.

INI sections are used to define styles, **[styles.mystyle]** defines a style called *mystle*. Within a section styles can be defined as:

```
[FORMAT].[PORTION].[SELECTOR].[ELEMENT] = [VALUE]
```

FORMAT
: Either **console** or **html**

PORTION
: Part of the key−value the style applies to: **key** to indicate the key part of a key−value, **value** to indicate the value part of a key−value, **flag** to indicate a single flag or **delim** to indicate delimiters such as parentheses, brackets, etc.

SELECTOR
: Select the key−value the style applies to: **highlighted** to indicate highlighted key−values, **type.<type>** to indicate certain types such as **IPAddress** or **EthMask** or **<keyname>** to select a particular key name.

ELEMENT
: Select the style element to modify: **color** or **underline** (only for **console** format).

VALUE
: Ether a color hex, other color names defined in the rich python library (-*https://rich.readthedocs.io/en/stable/appendix/colors.html*) or **true** if the element is **underline**.

A default configuration file is shipped with **ovs−flowviz** and its path is printed in the **−−help** output. A detailed description of the syntax alongside some examples are available there.

FILTERING SYNTAX

ovs−flowviz provides rich highlighting and filtering. The special command **ovs−flowviz filter** dumps the filtering syntax:

```
$ ovs-flowviz filter
Filter Syntax
*************

   [! | not ] {key}[[.subkey]...] [OPERATOR] {value})] [LOGICAL OPERATOR] ...

  Comparison operators:
      =   equality
      <   less than
      >   more than
      ~=  masking (valid for IP and Ethernet fields)

  Logical operators:
      !{expr}:  NOT
      {expr} && {expr}: AND
      {expr} || {expr}: OR

  Matches and flow metadata:
      To compare against a match or info field, use the field directly, e.g:
          priority=100
          n_bytes>10
      Use simple keywords for flags:
          tcp and ip_src=192.168.1.1

  Actions:
      Actions values might be dictionaries, use subkeys to access individual
      values, e.g:
          output.port=3
      Use simple keywords for flags
          drop

  Examples of valid filters:
      nw_addr~=192.168.1.1 && (tcp_dst=80 || tcp_dst=443)
      arp=true && !arp_tsa=192.168.1.1
      n_bytes>0 && drop=true
```

Example expressions:

```
n_bytes > 0 and drop
nw_src~=192.168.1.1 or arp.tsa=192.168.1.1
! tcp && output.port=2
```

EXAMPLES

Print OpenFlow flows sorted by cookie adding OVN data to each one:

```
$ ovs-flowviz -i flows.txt openflow cookie --ovn-detrace
```

Print OpenFlow logical structure, showing the flows and heat−map:

```
$ ovs-flowviz -i flows.txt openflow logic --show-flows --heat-map
```

Display OpenFlow flows in HTML format with “light” style and highlight drops:

```
$ ovs-flowviz -i flows.txt --style "light" --highlight "n_packets > 0 and drop" o
```

Display the datapath flows in an interactive graphviz + HTML view:

```
$ ovs-flowviz -i flows.txt datapath graph --html > flows.html
```

Display the datapath flow trees that lead to packets being sent to port 10:

```
$ ovs-flowviz -i flows.txt --filter "output.port=10" datapath tree
```

AUTHOR

The Open vSwitch Development Community

COPYRIGHT