Soft Patch Panel Documentation

Release 19.11

Jul 01, 2020

Contents

CHAPTER 1

Overview

Soft Patch Panel (SPP) is a DPDK application for providing switching functionality for Service Function Chaining in NFV (Network Function Virtualization).

Fig. 1.1: SPP overview

With SPP, user is able to configure network easily and dynamically via simple patch panel like interface.

The goal of SPP is to easily interconnect NFV applications via high thoughput network interfaces provided by DPDK and change configurations of resources dynamically to applications to build pipelines.

CHAPTER 2

Design

2.1 Soft Patch Panel

SPP is composed of several DPDK processes and controller processes for connecting each of client processes with high-throughput path of DPDK. Fig. 2.1 shows SPP processes and client apps for describing overview of design of SPP. In this diagram, solid line arrows describe a data path for packet forwarding and it can be configured from controller via command messaging of blue dashed line arrows.

Fig. 2.1: Overview of design of SPP

In terms of DPDK processes, SPP is derived from DPDK's multi-process sample application and it consists of a primary process and multiple secondary processes. SPP primary process is responsible for resource management, for example, initializing ports, mbufs or shared memory. On the other hand, secondary processes of `spp_nfv` are working for forwarding [1].

2.1.1 Reference

- [1] Implementation and Testing of Soft Patch Panel

2.2 SPP Controller

SPP is controlled from python based management framework. It consists of front-end CLI and back-end server process. SPP's front-end CLI provides a patch panel like interface for users. This CLI process parses user input and sends request to the back-end via REST APIs. It means that the back-end server process accepts requests from other than CLI. It enables developers to implement control interface such as GUI, or plugin for other framework. networking-spp is a Neutron ML2 plugin for using SPP with OpenStack. By using networking-spp and doing some of extra tunings for optimization, you can deploy high-performance NFV services on OpenStack [1].

2.2.1 spp-ctl

`spp-ctl` is designed for managing SPP from several controllers via REST-like APIs for users or other applications. It is implemented to be simple and it is stateless. Basically, it only converts a request into a command of SPP process and forward it to the process without any of syntax or lexical checking.

There are several usecases where SPP is managed from other process without user inputs. For example, you need a intermediate process if you think of using SPP from a framework, such as OpenStack. networking-spp is a Neutron ML2 plugin for SPP and *spp-agent* works as a SPP controller.

As shown in Fig. 2.2, `spp-ctl` behaves as a TCP server for SPP primary and secondary processes, and REST API server for client applications. It should be launched in advance to setup connections with other processes. `spp-ctl` uses three TCP ports for primary, secondaries and clients. The default port numbers are `5555`, `6666` and `7777`.

Fig. 2.2: Spp-ctl as a REST API server

`spp-ctl` accepts multiple requests at the same time and serializes them by using bottle which is simple and well known as a web framework and eventlet for parallel processing.

2.2.2 SPP CLI

SPP CLI is a user interface for managing SPP and implemented as a client of `spp-ctl`. It provides several kinds of command for inspecting SPP processes, changing path configuration or showing statistics of packets. However, you do not need to use SPP CLI if you use `netowrking-spp` or other client applications of `spp-ctl`. SPP CLI is one of them.

From SPP CLI, user is able to configure paths as similar as patch panel like manner by sending commands to each of SPP secondary processes. `patch phy:0 ring:0` is to connect two ports, `phy:0` and `ring:0`.

As described in *Getting Started* guide, SPP CLI is able to communicate several `spp-ctl` to support multiple nodes configuration.

2.2.3 Reference

- [1] Integrating OpenStack with DPDK for High Performance Applications

2.3 SPP Primary

SPP is originally derived from Client-Server Multi-process Example of Multi-process Sample Application in DPDK's sample applications. `spp_primary` is a server for other secondary processes and basically working same as described in "How the Application Works" section of the sample application.

However, there are some differences between `spp_primary` and the server process of the sample application. `spp_primary` has no limitation of the number of secondary processes. It does not work for packet forwaring without in some usecases, but just provide rings and memory pools for secondary processes.

Primary process supports `rte_flow` of DPDK for hardware offloading. Packet distribution based on dst MAC address and/or VLAN ID is supported. Entag/detag of VLAN is also supported.

2.3.1 Master and Worker Threads

In SPP, Both of primary and secondary processes consist of master thread and worker thread as slave. Master thread is for accepting commands from a user for doing task, and running on a master lcore. On the other hand, slave thread is for packet forwarding or other process specific jobs as worker, and running on slave lcore. Only slave thread requires dedicated core for running in pole mode, and launched from `rte_eal_remote_launch()` or `rte_eal_mp_remote_launch()`.

`spp_primary` is able to run with or without worker thread selectively, and requires at least one lcore for server process. Using worker thread or not depends on your usecases. `spp_primary` provides two types of workers currently.

2.3.2 Worker Types

There are two types of worker thread in `spp_primary`. First one is is forwarder thread, and second one is monitor thread.

As default, `spp_primary` runs packet forwarder if two or more lcores are given while launching the process. Behavior of this forwarder is same as `spp_nfv` described in the next section. This forwarder provides features for managing ports, patching them and forwarding packets between ports. It is useful for very simple usecase in which only few ports are patched and no need to do forwarding packets in parallel with several processes.

Note: In DPDK v18.11 or later, some of PMDs, such as vhost, do not work for multi-process application. It means that packets cannot forwarded to a VM or container via secondary process in SPP. In this case, you use forwarder in `spp_primary`.

Another type is monitor for displaying status of `spp_primary` in which statistics of RX and TX packets on each of physical ports and ring ports are shown periodically in terminal `spp_primary` is launched. Although statistics can be referred in SPP CLI by using `pri; status` command, running monitor thread is useful if you always watch statistics.

2.4 SPP Secondary

SPP secondary process is a worker process in client-server multp-process application model. Basically, the role of secondary process is to connenct each of application running on host, containers or VMs for packet forwarding. Spp secondary process forwards packets from source port to destination port with DPDK's high-performance forwarding mechanizm. In other word, it behaves as a cable to connect two patches ports.

All of secondary processes are able to attach ring PMD and vhost PMD ports for sending or receiving packets with other processes. Ring port is used to communicate with a process running on host or container if it is implemented as secondary process to access shared ring

memory. Vhost port is used for a process on container or VM and implemented as primary process, and no need to access shared memory of SPP primary.

In addition to the basic forwarding, SPP secondary process provides several networking features. One of the typical example is packet cauture. `spp_nfv` is the simplest SPP secondary and used to connect two of processes or other feature ports including PCAP PMD port. PCAP PMD is to dump packets to a file or retrieve from.

There are more specific or funcional features than `spp_nfv`. `spp_vf` is a simple pseudo SR-IOV feature for classifying or merging packets. `spp_mirror` is to duplicate incoming packets to several destination ports.

2.4.1 spp_nfv

`spp_nfv` is the simplest SPP secondary to connect two of processes or other feature ports. Each of `spp_nfv` processes has a list of entries including source and destination ports, and forwards packets by referring the list. It means that one `spp_nfv` might have several forwarding paths, but throughput is gradually decreased if it has too much paths. This list is implemented as an array of `port` structure and named `ports_fwd_array`. The index of `ports_fwd_array` is the same as unique port ID.

```
struct port {
  int in_port_id;
  int out_port_id;
  ...
};
...

/* ports_fwd_array is an array of port */
static struct port ports_fwd_array[RTE_MAX_ETHPORTS];
```

Fig. 2.3 describes an example of forwarding between ports. In this case, `spp_nfv` is responsible for forwarding from `port#0` to `port#2`. You notice that each of `out_port` entry has the destination port ID.

Fig. 2.3: Forwarding by referring ports_fwd_array

`spp_nfv` consists of main thread and worker thread to update the entry while running the process. Main thread is for waiting user command for updating the entry. Worker thread is for dedicating packet forwarding. Fig. 2.4 describes tasks in each of threads. Worker thread is launched from main thread after initialization. In worker thread, it starts forwarding if user send forward command and main thread accepts it.

Fig. 2.4: Main thread and worker thread in spp_nfv

2.4.2 spp_vf

`spp_vf` provides a SR-IOV like network feature.

`spp_vf` forwards incoming packets to several destination VMs by referring MAC address like as a Virtual Function (VF) of SR-IOV.

`spp_vf` is a multi-process and multi-thread application. Each of `spp_vf` has one manager thread and worker threads called as components. The manager thread provides a function for parsing a command and creating the components. The component threads have its own multiple components, ports and classifier tables including Virtual MAC address. There are three types of components, `forwarder`, `merger` and `classifier`.

This is an example of network configuration, in which one `classifier`, one merger and four forwarders are running in `spp_vf` process for two destinations of vhost interface. Incoming packets from rx on host1 are sent to each of vhosts of VM by looking up destination MAC address in the packet.

Fig. 2.5: Classification of spp_vf for two VMs

Forwarder

Simply forwards packets from rx to tx port. Forwarder does not start forwarding until when at least one rx and one tx are added.

Merger

Receives packets from multiple rx ports to aggregate packets and sends to a desctination port. Merger does not start forwarding until when at least two rx and one tx are added.

Classifier

Sends packets to multiple tx ports based on entries of MAC address and destination port in a classifier table. This component also supports VLAN tag.

For VLAN addressing, classifier has other tables than defalut. Classifier prepares tables for each of VLAN ID and decides which of table is referred if TPID (Tag Protocol Indetifier) is included in a packet and equals to 0x8100 as defined in IEEE 802.1Q standard. Classifier does not start forwarding until when at least one rx and two tx are added.

2.4.3 spp_mirror

`spp_mirror` is an implementation of TaaS as a SPP secondary process for port mirroring. TaaS stands for TAP as a Service. The keyword `mirror` means that it duplicates incoming packets and forwards to additional destination.

Mirror

`mirror` component has one `rx` port and two `tx` ports. Incoming packets from `rx` port are duplicated and sent to each of `tx` ports.

Fig. 2.6: Spp_mirror component

In general, copying packet is time-consuming because it requires to make a new region on memory space. Considering to minimize impact for performance, `spp_mirror` provides a

choice of copying methods, `shallowocopy` or `deepcopy`. The difference between those methods is `shallowocopy` does not copy whole of packet data but share without header actually. `shallowcopy` is to share mbuf between packets to get better performance than `deepcopy`, but it should be used for read only for the packet.

Note: `shallowcopy` calls `rte_pktmbuf_clone()` internally and `deepcopy` create a new mbuf region.

You should choose `deepcopy` if you use VLAN feature to make no change for original packet while copied packet is modified.

2.4.4 spp_pcap

SPP provides a connectivity between VM and NIC as a virtual patch panel. However, for more practical use, operator and/or developer needs to capture packets. For such use, spp_pcap provides packet capturing feature from specific port. It is aimed to capture up to 10Gbps packets.

`spp_pcap` is a SPP secondary process for capturing packets from specific `port`. Fig. 2.7 shows an overview of use of `spp_pcap` in which `spp_pcap` process receives packets from `phy:0` for capturing.

Note: `spp_pcap` supports only two types of ports for capturing, `phy` and `ring`, currently.

Fig. 2.7: Overview of spp_pcap

`spp_pcap` cosisits of main thread, `receiver` thread and one or more `wirter` threads. As design policy, the number of `receiver` is fixed to 1 because to make it simple and it is enough for task of receiving. `spp_pcap` requires at least three lcores, and assign to from master, `receiver` and then the rest of `writer` threads respectively.

Incoming packets are received by `receiver` thread and transferred to `writer` threads via ring buffers between threads.

Several `writer` work in parallel to store packets as files in LZ4 format. You can capture a certain amount of heavy traffic by using much `writer` threads.

Fig. 2.8 shows an usecase of `spp_pcap` in which packets from `phy:0` are captured by using three `writer` threads.

Fig. 2.8: spp_pcap internal structure

2.5 Implementation

This section describes topics of implementation of SPP processes.

2.5.1 spp_nfv

`spp_nfv` is a DPDK secondary process and communicates with primary and other peer processes via TCP sockets or shared memory. `spp_nfv` consists of several threads, main thread for maanging behavior of `spp_nfv` and worker threads for packet forwarding.

As initialization of the process, it calls `rte_eal_init()`, then specific initialization functions for resources of `spp_nfv` itself.

After initialization, main thread launches worker threads on each of given slave lcores with `rte_eal_remote_launch()`. It means that `spp_nfv` requires two lcores at least. Main thread starts to accept user command after all of worker threads are launched.

Initialization

In main funciton, `spp_nfv` calls `rte_eal_init()` first as other DPDK applications, `forward_array_init()` and `port_map_init()` for initializing port forward array which is a kind of forwarding table.

```
int
main(int argc, char *argv[])
{
        ....

        ret = rte_eal_init(argc, argv);
        if (ret < 0)
                return -1;
        ....

        /* initialize port forward array*/
        forward_array_init();
        port_map_init();
        ....
```

Port forward array is implemented as an array of `port` structure. It consists of RX, TX ports and its forwarding functions, `rte_rx_burst()` and `rte_tx_burst()` actually. Each of ports are identified with unique port ID. Worker thread iterates this array and forward packets from RX port to TX port.

```
/* src/shared/common.h */

struct port {
    uint16_t in_port_id;
    uint16_t out_port_id;
    uint16_t (*rx_func)(uint16_t, uint16_t, struct rte_mbuf **, uint16_t);
    uint16_t (*tx_func)(uint16_t, uint16_t, struct rte_mbuf **, uint16_t);
};
```

Port map is another kind of structure for managing its type and statistics. Port type for indicating PMD type, for example, ring, vhost or so. Statistics is used as a counter of packet forwarding.

```
/* src/shared/common.h */

struct port_map {
        int id;
        enum port_type port_type;
        struct stats *stats;
```

(continues on next page)

(continued from previous page)

```
        struct stats default_stats;
};
```

Final step of initialization is setting up memzone. In this step, `spp_nfv` just looks up memzone of primary process as a secondary.

```
/* set up array for port data */
if (rte_eal_process_type() == RTE_PROC_SECONDARY) {
        mz = rte_memzone_lookup(MZ_PORT_INFO);
        if (mz == NULL)
                rte_exit(EXIT_FAILURE,
                        "Cannot get port info structure\n");
        ports = mz->addr;
```

Launch Worker Threads

Worker threads are launched with `rte_eal_remote_launch()` from main thread. `RTE_LCORE_FOREACH_SLAVE` is a macro for traversing slave lcores while incrementing `lcore_id` and `rte_eal_remote_launch()` is a function for running a function on worker thread.

```
lcore_id = 0;
RTE_LCORE_FOREACH_SLAVE(lcore_id) {
        rte_eal_remote_launch(main_loop, NULL, lcore_id);
}
```

In this case, `main_loop` is a starting point for calling task of worker thread `nfv_loop()`.

```
static int
main_loop(void *dummy __rte_unused)
{
        nfv_loop();
        return 0;
}
```

Parsing User Command

After all of worker threads are launched, main threads goes into while loop for waiting user command from SPP controller via TCP connection. If receiving a user command, it simply parses the command and make a response. It terminates the while loop if it receives `exit` command.

```
while (on) {
        ret = do_connection(&connected, &sock);
        ....
        ret = do_receive(&connected, &sock, str);
        ....
        flg_exit = parse_command(str);
        ....
        ret = do_send(&connected, &sock, str);
        ....
}
```

`parse_command()` is a function for parsing user command as named. There are several commnads for `spp_nfv` as described in *Secondary Commands*. Command from controller is a simple plain text and action for the command is decided with the first token of the command.

```
static int
parse_command(char *str)
{
        ....

        if (!strcmp(token_list[0], "status")) {
                RTE_LOG(DEBUG, SPP_NFV, "status\n");
                memset(str, '\0', MSG_SIZE);
        ....

                } else if (!strcmp(token_list[0], "add")) {
                RTE_LOG(DEBUG, SPP_NFV, "Received add command\n");
                if (do_add(token_list[1]) < 0)
                        RTE_LOG(ERR, SPP_NFV, "Failed to do_add()\n");

        } else if (!strcmp(token_list[0], "patch")) {
                RTE_LOG(DEBUG, SPP_NFV, "patch\n");
        ....
}
```

For instance, if the first token is `add`, it calls `do_add()` with given tokens and adds port to the process.

2.5.2 spp_vf

This section describes implementation of key features of `spp_vf`.

`spp_vf` consists of master thread and several worker threads, `forwarder`, `classifier` or `merger`, as slaves. For classifying packets, `spp_vf` has a worker thread named `classifier` and a table for registering MAC address entries.

Initialization

In master thread, data of classifier and VLAN features are initialized after `rte_eal_init()` is called. Port capability is a set of data for describing VLAN features. Then, each of worker threads are launched with `rte_eal_remote_launch()` on assigned lcores..

```
 /* spp_vf.c */

 ret = rte_eal_init(argc, argv);

 /* skipping lines ... */

 /* Start worker threads of classifier and forwarder */
unsigned int lcore_id = 0;
RTE_LCORE_FOREACH_SLAVE(lcore_id) {
        rte_eal_remote_launch(slave_main, NULL, lcore_id);
}
```

Slave Main

Main function of worker thread is defined as `slave_main()` which is called from `rte_eal_remote_launch()`. Behavior of worker thread is decided in while loop in this function. If lcore status is not `SPPWK_LCORE_RUNNING`, worker thread does nothing. On the other hand, it does packet forwarding with or without classifying. It classifies incoming packets if component type is `SPPWK_TYPE_CLS`, or simply forwards packets.

```
/* spp_vf.c */

while ((status = spp_get_core_status(lcore_id)) !=
    SPPWK_LCORE_REQ_STOP) {
    if (status != SPPWK_LCORE_RUNNING)
        continue;

    /* skipping lines ... */

    /* It is for processing multiple components. */
    for (cnt = 0; cnt < core->num; cnt++) {
    /* Component classification to call a function. */
    if (spp_get_component_type(core->id[cnt]) ==
                SPPWK_TYPE_CLS) {
        /* Component type for classifier. */
        ret = classify_packets(core->id[cnt]);
        if (unlikely(ret != 0))
                break;
    } else {
        /* Component type for forward or merge. */
            ret = forward_packets(core->id[cnt]);
            if (unlikely(ret != 0))
                break;
        }
    }
```

Data structure of classifier

Classifier has a set of attributes for classification as struct `mac_classifier`, which consists of a table of MAC addresses, number of classifying ports, indices of ports and default index of port. Clasifier table is implemented as hash of struct `rte_hash`.

```
/* shared/secondary/spp_worker_th/vf_deps.h */

/* Classifier for MAC addresses. */
struct mac_classifier {
    struct rte_hash *cls_tbl;  /* Hash table for MAC classification. */
    int nof_cls_ports;  /* Num of ports classified validly. */
    int cls_ports[RTE_MAX_ETHPORTS];  /* Ports for classification. */
    int default_cls_idx;  /* Default index for classification. */
};
```

Classifier itself is defined as a struct `cls_comp_info`. There are several attributes in this struct including `mac_classifier` or `cls_port_info` or so. `cls_port_info` is for defining a set of attributes of ports, such as interface type, device ID or packet data.

```
/* shared/secondary/spp_worker_th/vf_deps.h */

/* classifier component information */
```

(continues on next page)

(continued from previous page)

```
struct cls_comp_info {
    char name[STR_LEN_NAME];  /* component name */
    int mac_addr_entry;  /* mac address entry flag */
    struct mac_classifier *mac_clfs[NOF_VLAN];  /* classifiers per VLAN. */
    int nof_tx_ports;  /* Number of TX ports info entries. */
    /* Classifier has one RX port and several TX ports. */
    struct cls_port_info rx_port_i;  /* RX port info classified. */
    struct cls_port_info tx_ports_i[RTE_MAX_ETHPORTS];  /* TX info. */
};

/* Attirbutes of port for classification. */
struct cls_port_info {
    enum port_type iface_type;
    int iface_no;   /* Index of ports handled by classifier. */
    int iface_no_global;  /* ID for interface generated by spp_vf */
    uint16_t ethdev_port_id;  /* Ethdev port ID. */
    uint16_t nof_pkts;  /* Number of packets in pkts[]. */
    struct rte_mbuf *pkts[MAX_PKT_BURST];  /* packets to be classified. */
};
```

Classifying the packet

If component type is `SPPWK_TYPE_CLS`, worker thread behaves as a classifier, so component calls `classify_packets()`. In this function, packets from RX port are received with `sppwk_eth_vlan_rx_burst()` which is derived from `rte_eth_rx_burst()` for adding or deleting VLAN tags. Received packets are classified with `classify_packet()`.

```
/* classifier.c */

n_rx = sppwk_eth_vlan_rx_burst(clsd_data_rx->ethdev_port_id, 0,
    rx_pkts, MAX_PKT_BURST);

/* skipping lines ... */

classify_packet(rx_pkts, n_rx, cmp_info, clsd_data_tx);
```

Packet processing in forwarder and merger

Configuration data for forwarder and merger is stored as structured tables `forward_rxtx`, `forward_path` and `forward_info`. The `forward_rxtx` has two member variables for expressing the port to be sent(tx) and to be receive(rx), `forward_path` has member variables for expressing the data path. Like as `mac_classifier`, `forward_info` has two tables, one is for updating by commands, the other is for looking up to process packets.

```
/* forwarder.c */
/* A set of port info of rx and tx */
struct forward_rxtx {
        struct spp_port_info rx; /* rx port */
        struct spp_port_info tx; /* tx port */
};

/* Information on the path used for forward. */
struct forward_path {
        char name[STR_LEN_NAME];  /* Component name */
        volatile enum sppwk_worker_type wk_type;
```

(continues on next page)

(continued from previous page)

```
        int nof_rx;  /* Number of RX ports */
        int nof_tx;  /* Number of TX ports */
        struct forward_rxtx ports[RTE_MAX_ETHPORTS];  /* Set of RX and TX */
};

/* Information for forward. */
struct forward_info {
        volatile int ref_index; /* index to reference area */
        volatile int upd_index; /* index to update area    */
        struct forward_path path[SPP_INFO_AREA_MAX];
                                /* Information of data path */
};
```

L2 Multicast Support

`spp_vf` supports multicast for resolving ARP requests. It is implemented as `handle_l2multicast_packet()` and called from `classify_packet()` for incoming multicast packets.

```
/* classify_packet() in classifier.c */

/* L2 multicast(include broadcast) ? */
if (unlikely(is_multicast_ether_addr(&eth->d_addr))) {
        RTE_LOG(DEBUG, SPP_CLASSIFIER_MAC,
                        "multicast mac address.\n");
        handle_l2multicast_packet(rx_pkts[i],
                        classifier_info,
                        classified_data);
        continue;
}
```

Packets are cloned with `rte_mbuf_refcnt_update()` for distributing multicast packets.

```
/* classifier.c */

handle_l2multicast_packet(struct rte_mbuf *pkt,
        struct cls_comp_info *cmp_info,
        struct cls_port_info *clsd_data)
{
        int i;
        struct mac_classifier *mac_cls;
        uint16_t vid = get_vid(pkt);
        int gen_def_clsd_idx = get_general_default_classified_index(cmp_info);
        int n_act_clsd;

        /* skipping lines... */

        rte_mbuf_refcnt_update(pkt, (int16_t)(n_act_clsd - 1));
```

Two phase update for forwarding

Update of netowrk configuration in `spp_vf` is done in a short period of time, but not so short considering the time scale of packet forwarding. It might forward packets before the updating is completed possibly. To avoid such kind of situation, `spp_vf` has two phase update mechanism. Status info is referred from forwarding process after the update is completed.

```
int
flush_cmd(void)
{
    int ret;
    int *p_change_comp;
    struct sppwk_comp_info *p_comp_info;
    struct cancel_backup_info *backup_info;

    sppwk_get_mng_data(NULL, &p_comp_info, NULL, NULL, &p_change_comp,
            &backup_info);

    ret = update_port_info();
    if (ret < SPPWK_RET_OK)
        return ret;

    update_lcore_info();

    ret = update_comp_info(p_comp_info, p_change_comp);

    backup_mng_info(backup_info);
    return ret;
}
```

2.5.3 spp_mirror

This section describes implementation of `spp_mirror`. It consists of master thread and several worker threads for duplicating packets.

Slave Main

Main function of worker thread is defined as `slave_main()` in which for duplicating packets is `mirror_proc()` on each of lcores.

```
for (cnt = 0; cnt < core->num; cnt++) {

    ret = mirror_proc(core->id[cnt]);
    if (unlikely(ret != 0))
        break;
}
```

Mirroring Packets

Worker thread receives and duplicate packets. There are two modes of copying packets, `shallowcopy` and `deepcopy`. Deep copy is for duplicating whole of packet data, but less performance than shallow copy. Shallow copy duplicates only packet header and body is not shared among original packet and duplicated packet. So, changing packet data affects both of original and copied packet.

You can configure using which of modes in Makefile. Default mode is `shallowcopy`. If you change the mode to `deepcopy`, comment out this line of CFLAGS.

```
# Default mode is shallow copy.
CFLAGS += -DSPP_MIRROR_SHALLOWCOPY
```

This code is a part of `mirror_proc()`. In this function, `rte_pktmbuf_clone()` is just called if in shallow copy mode, or create a new packet with `rte_pktmbuf_alloc()` for duplicated packet if in deep copy mode.

```
            for (cnt = 0; cnt < nb_rx; cnt++) {
                    org_mbuf = bufs[cnt];
                    rte_prefetch0(rte_pktmbuf_mtod(org_mbuf, void *));
#ifdef SPP_MIRROR_SHALLOWCOPY
                    /* Shallow Copy */
        copybufs[cnt] = rte_pktmbuf_clone(org_mbuf,
                                                g_mirror_pool);

#else
                    struct rte_mbuf *mirror_mbuf = NULL;
                    struct rte_mbuf **mirror_mbufs = &mirror_mbuf;
                    struct rte_mbuf *copy_mbuf = NULL;
                    /* Deep Copy */
                    do {
                            copy_mbuf = rte_pktmbuf_alloc(g_mirror_pool);
                            if (unlikely(copy_mbuf == NULL)) {
                                    rte_pktmbuf_free(mirror_mbuf);
                                    mirror_mbuf = NULL;
                                    RTE_LOG(INFO, MIRROR,
                                            "copy mbuf alloc NG!\n");
                                    break;
                            }

                            copy_mbuf->data_off = org_mbuf->data_off;
                            ...
                            copy_mbuf->packet_type = org_mbuf->packet_type;

                            rte_memcpy(rte_pktmbuf_mtod(copy_mbuf, char *),
                                    rte_pktmbuf_mtod(org_mbuf, char *),
                                    org_mbuf->data_len);

                            *mirror_mbufs = copy_mbuf;
                            mirror_mbufs = &copy_mbuf->next;
                    } while ((org_mbuf = org_mbuf->next) != NULL);
        copybufs[cnt] = mirror_mbuf;

#endif /* SPP_MIRROR_SHALLOWCOPY */
            }
    if (cnt != 0)
                    nb_tx2 = spp_eth_tx_burst(tx->dpdk_port, 0,
                            copybufs, cnt);
```

2.5.4 spp_pcap

This section describes implementation of `spp_pcap`.

Slave Main

In `slave_main()`, it calls `pcap_proc_receive()` if thread type is receiver, or `pcap_proc_write()` if the type is writer.

```
/* spp_pcap.c */
```

(continues on next page)

(continued from previous page)

```
while ((status = spp_get_core_status(lcore_id)) !=
                SPP_CORE_STOP_REQUEST) {

        if (pcap_info->type == TYPE_RECIVE)
                ret = pcap_proc_receive(lcore_id);
        else
                ret = pcap_proc_write(lcore_id);
        }
}
```

Receiving Pakcets

`pcap_proc_receive()` is for receiving packets with `rte_eth_rx_burst` and sending the packets to the writer thread via ring memory by using `rte_ring_enqueue_bulk()`.

```
/* spp_pcap.c */

rx = &g_pcap_option.port_cap;
nb_rx = rte_eth_rx_burst(rx->ethdev_port_id, 0, bufs, MAX_PCAP_BURST);

/* Forward to ring for writer thread */
nb_tx = rte_ring_enqueue_burst(write_ring, (void *)bufs, nb_rx, NULL);
```

Writing Packet

`pcap_proc_write()` is for capturing packets to a file. The captured file is compressed with LZ4 which is a lossless compression algorithm and providing compression speed > 500 MB/s per core.

```
nb_rx =  rte_ring_dequeue_bulk(read_ring, (void *)bufs, MAX_PKT_BURST,
                                                                NULL);
for (buf = 0; buf < nb_rx; buf++) {
        mbuf = bufs[buf];
        rte_prefetch0(rte_pktmbuf_mtod(mbuf, void *));
        if (compress_file_packet(&g_pcap_info[lcore_id], mbuf)
                                                != SPPWK_RET_OK) {
                RTE_LOG(ERR, PCAP, "capture file write error: "
                        "%d (%s)\n", errno, strerror(errno));
                ret = SPPWK_RET_NG;
                info->status = SPP_CAPTURE_IDLE;
                compress_file_operation(info, CLOSE_MODE);
                break;
        }
}
for (buf = nb_rx; buf < nb_rx; buf++)
        rte_pktmbuf_free(bufs[buf]);
```

CHAPTER 3

Getting Started Guide

3.1 Setup

This documentation is described for following distributions.

- Ubuntu 16.04 and 18.04
- CentOS 7.6 (not fully supported)

3.1.1 Reserving Hugepages

Hugepages should be enabled for running DPDK application. Hugepage support is to reserve large amount size of pages, 2MB or 1GB per page, to less TLB (Translation Lookaside Buffers) and to reduce cache miss. Less TLB means that it reduce the time for translating virtual address to physical.

How to configure reserving hugepages is different between 2MB or 1GB. In general, 1GB is better for getting high performance, but 2MB is easier for configuration than 1GB.

1GB Hugepage

For using 1GB page, hugepage setting is activated while booting system. It must be defined in boot loader configuration, usually it is `/etc/default/grub`. Add an entry of configuration of the size and the number of pages.

Here is an example for Ubuntu, and almost the same as CentOS. The points are that `hugepagesz` is for the size and `hugepages` is for the number of pages. You can also configure `isolcpus` in grub setting for improving performance as described in *Performance Optimizing*.

```
# /etc/default/grub
GRUB_CMDLINE_LINUX="default_hugepagesz=1G hugepagesz=1G hugepages=8"
```

For Ubuntu, you should run `update-grub` for updating `/boot/grub/grub.cfg` after editing to update grub's config file, or this configuration is not activated.

```
# Ubuntu
$ sudo update-grub
Generating grub configuration file ...
```

Or for CentOS7, you use `grub2-mkconfig` instead of `update-grub`. In this case, you should give the output file with `-o` option. The output path might be different, so you should find your correct `grub.cfg` by yourself.

```
# CentOS
$ sudo grub2-mkconfig -o /boot/efi/EFI/centos/grub.cfg
```

Note: 1GB hugepages might not be supported on your hardware. It depends on that CPUs support 1GB pages or not. You can check it by referring `/proc/cpuinfo`. If it is supported, you can find `pdpe1gb` in the `flags` attribute.

```
$ cat /proc/cpuinfo | grep pdpe1gb
flags           : fpu vme ... pdpe1gb ...
```

2MB Hugepage

For using 2MB page, you can activate hugepages while booting or at anytime after system is booted. Define hugepages setting in `/etc/default/grub` to activate it while booting, or overwrite the number of 2MB hugepages as following.

```
$ echo 1024 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

In this case, 1024 pages of 2MB, totally 2048 MB, are reserved.

3.1.2 Mount hugepages

Make the memory available for using hugepages from DPDK.

```
$ mkdir /mnt/huge
$ mount -t hugetlbfs nodev /mnt/huge
```

It is also available while booting by adding a configuration of mount point in `/etc/fstab`. The mount point for 2MB or 1GB can be made permanently accross reboot. For 2MB, it is no need to declare the size of hugepages explicity.

```
# /etc/fstab
nodev /mnt/huge hugetlbfs defaults 0 0
```

For 1GB, the size of hugepage `pagesize` must be specified.

```
# /etc/fstab
nodev /mnt/huge_1GB hugetlbfs pagesize=1GB 0 0
```

3.1.3 Disable ASLR

SPP is a DPDK multi-process application and there are a number of limitations .

Address-Space Layout Randomization (ASLR) is a security feature for memory protection, but may cause a failure of memory mapping while starting multi-process application as discussed in dpdk-dev .

ASLR can be disabled by assigning `kernel.randomize_va_space` to `0`, or be enabled by assigning it to `2`.

```
# disable ASLR
$ sudo sysctl -w kernel.randomize_va_space=0

# enable ASLR
$ sudo sysctl -w kernel.randomize_va_space=2
```

You can check the value as following.

```
$ sysctl -n kernel.randomize_va_space
```

3.1.4 Using Virtual Machine

SPP provides vhost interface for inter VM communication. You can use any of DPDK supported hypervisors, but this document describes usecases of qemu and libvirt.

Server mode v.s. Client mode

For using vhost, vhost port should be created before VM is launched in server mode, or SPP is launched in client mode to be able to create vhost port after VM is launched.

Client mode is optional and supported in qemu 2.7 or later. For using this mode, launch secondary process with `--vhost-client`. Qemu creates socket file instead of secondary process. It means that you can launch a VM before secondary process create vhost port.

Libvirt

If you use libvirt for managing virtual machines, you might need some additional configurations.

To have access to resources with your account, update and activate user and group parameters in `/etc/libvirt/qemu.conf`. Here is an example.

```
# /etc/libvirt/qemu.conf

user = "root"
group = "root"
```

For using hugepages with libvirt, change `KVM_HUGEPAGES` from 0 to 1 in `/etc/default/qemu-kvm`.

```
# /etc/default/qemu-kvm

KVM_HUGEPAGES=1
```

Change grub config as similar to *Reserving Hugepages*. You can check hugepage settings as following.

```
$ cat /proc/meminfo | grep -i huge
AnonHugePages:      2048 kB
HugePages_Total:      36            #       /etc/default/grub
HugePages_Free:       36
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    1048576 kB         #       /etc/default/grub

$ mount | grep -i huge
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,...,nsroot=/)
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime)
hugetlbfs-kvm on /run/hugepages/kvm type hugetlbfs (rw,...,gid=117)
hugetlb on /run/lxcfs/controllers/hugetlb type cgroup (rw,...,nsroot=/)
```

Finally, you umount default hugepages.

```
$ sudo umount /dev/hugepages
```

Trouble Shooting

You might encounter a permission error while creating a resource, such as a socket file under `tmp/`, because of AppArmor.

You can avoid this error by editing `/etc/libvirt/qemu.conf`.

```
# Set security_driver to "none"
$sudo vi /etc/libvirt/qemu.conf
...
security_driver = "none"
...
```

Restart libvirtd to activate this configuration.

```
$sudo systemctl restart libvirtd.service
```

Or, you can also avoid by simply removing AppArmor itself.

```
$ sudo apt-get remove apparmor
```

If you use CentOS, confirm that SELinux doesn't prevent for permission. SELinux is disabled simply by changing the configuration to `disabled`.

```
# /etc/selinux/config
SELINUX=disabled
```

Check your SELinux configuration.

```
$ getenforce
Disabled
```

3.1.5 Python 2 or 3 ?

Without SPP container tools, Python2 is not supported anymore. SPP container will also be updated to Python3.

3.1.6 Driver for Mellanox NIC

In case of using MLX5 NIC, you have to install driver. You can download driver from Mellanox's *SW/Drivers <https://www.mellanox.com/page/mlnx_ofed_matrix?mtag=linux_sw_drivers>*. The following example assumes that MLNX_OFED_LINUX-4.7-1.0.0.1-ubuntu18.04-x86_64.tgz is downloaded.

```
$cd MLNX_OFED_LINUX-4.7-1.0.0.1-ubuntu18.04-x86_64/
$sudo ./mlnxofedinstall --upstream-libs --dpdk --force
```

3.1.7 Reference

- [1] Use of Hugepages in the Linux Environment
- [2] Using Linux Core Isolation to Reduce Context Switches
- [3] Linux boot command line

3.2 Install DPDK and SPP

Before setting up SPP, you need to install DPDK. In this document, briefly described how to install and setup DPDK. Refer to DPDK documentation for more details. For Linux, see Getting Started Guide for Linux .

3.2.1 Required Packages

Installing packages for DPDK and SPP is almost the on Ubunu and CentOS, but names are different for some packages.

Ubuntu

To compile DPDK, it is required to install following packages.

```
$ sudo apt install libnuma-dev \
  libarchive-dev \
  build-essential
```

You also need to install linux-headers of your kernel version.

```
$ sudo apt install linux-headers-$(uname -r)
```

Some of SPP secondary processes depend on other libraries and you fail to compile SPP without installing them.

SPP provides libpcap-based PMD for dumping packet to a file or retrieve it from the file. `spp_nfv` and `spp_pcap` use `libpcap-dev` for packet capture. `spp_pcap` uses `liblz4-dev` and `liblz4-tool` to compress PCAP file.

```
$ sudo apt install libpcap-dev \
  liblz4-dev \
  liblz4-tool
```

`text2pcap` is also required for creating pcap file which is included in `wireshark`.

```
$ sudo apt install wireshark
```

CentOS

Before installing packages for DPDK, you should add IUS Community repositories with yum command.

```
$ sudo yum install https://centos7.iuscommunity.org/ius-release.rpm
```

To compile DPDK, required to install following packages.

```
$ sudo yum install numactl-devel \
  libarchive-devel \
  kernel-headers \
  kernel-devel
```

As same as Ubuntu, you should install additional packages because SPP provides libpcap-based PMD for dumping packet to a file or retrieve it from the file. `spp_nfv` and `spp_pcap` use `libpcap-dev` for packet capture. `spp_pcap` uses `liblz4-dev` and `liblz4-tool` to compress PCAP file. `text2pcap` is also required for creating pcap file which is included in `wireshark`.

```
$ sudo apt install libpcap-dev \
  libpcap \
  libpcap-devel \
  lz4 \
  lz4-devel \
  wireshark \
  wireshark-devel \
  libX11-devel
```

3.2.2 DPDK

Clone repository and compile DPDK in any directory.

```
$ cd /path/to/any
$ git clone http://dpdk.org/git/dpdk
```

Installing on Ubuntu and CentOS are almost the same, but required packages are just bit different.

PCAP is disabled by default in DPDK configuration. `CONFIG_RTE_LIBRTE_PMD_PCAP` and `CONFIG_RTE_PORT_PCAP` defined in config file `common_base` should be changed to `y` to enable PCAP.

```
# dpdk/config/common_base
CONFIG_RTE_LIBRTE_PMD_PCAP=y
...
CONFIG_RTE_PORT_PCAP=y
```

Compilation of igb_uio module is disabled by default in DPDK configuration. `CONFIG_RTE_EAL_IGB_UIO` defined in config file `common_base` should be changed to `y` to enable compilation of igb_uio.

```
# dpdk/config/common_base
CONFIG_RTE_EAL_IGB_UIO=y
```

If you use MLX5 NIC, `CONFIG_RTE_LIBRTE_MLX5_PMD` defined in config file `common_base` should be changed to `y`.

```
# dpdk/config/common_base
CONFIG_RTE_LIBRTE_MLX5_PMD=y
```

Compile DPDK with target environment.

```
$ cd dpdk
$ export RTE_SDK=$(pwd)
$ export RTE_TARGET=x86_64-native-linux-gcc  # depends on your env
$ make install T=$RTE_TARGET
```

3.2.3 Pyhton

Python3 and pip3 are also required because SPP controller is implemented in Pyhton3. Required packages can be installed from `requirements.txt`.

```
# Ubuntu
$ sudo apt install python3 \
  python3-pip
```

For CentOS, you need to specify minor version of python3. Here is an example of installing python3.6.

```
# CentOS
$ sudo yum install python36 \
  python36-pip
```

SPP provides `requirements.txt` for installing required packages of Python3. You might fail to run `pip3` without sudo on some environments.

```
$ pip3 install -r requirements.txt
```

For some environments, `pip3` might install packages under your home directory `$HOME/.local/bin` and you should add it to `$PATH` environment variable.

3.2.4 SPP

Clone SPP repository and compile it in any directory.

```
$ cd /path/to/any
$ git clone http://dpdk.org/git/apps/spp
$ cd spp
$ make  # Confirm that $RTE_SDK and $RTE_TARGET are set
```

If you use `spp_mirror` in deep copy mode, which is used for cloning whole of packet data for modification, you should change configuration of copy mode in Makefile of `spp_mirror` before. It is for copying full payload into a new mbuf. Default mode is shallow copy.

```
# src/mirror/Makefile
#CFLAGS += -Dspp_mirror_SHALLOWCOPY
```

Note: Before run make command, you might need to consider if using deep copy for cloning packets in `spp_mirror`. Comparing with shallow copy, it clones entire packet payload into a new mbuf and it is modifiable, but lower performance. Which of copy mode should be chosen depends on your usage.

3.2.5 Binding Network Ports to DPDK

Network ports must be bound to DPDK with a UIO (Userspace IO) driver. UIO driver is for mapping device memory to userspace and registering interrupts.

UIO Drivers

You usually use the standard `uio_pci_generic` for many use cases or `vfio-pci` for more robust and secure cases. Both of drivers are included by default in modern Linux kernel.

```
# Activate uio_pci_generic
$ sudo modprobe uio_pci_generic

# or vfio-pci
$ sudo modprobe vfio-pci
```

You can also use kmod included in DPDK instead of `uio_pci_generic` or `vfio-pci`.

```
$ sudo modprobe uio
$ sudo insmod kmod/igb_uio.ko
```

Binding Network Ports

Once UIO driver is activated, bind network ports with the driver. DPDK provides `usertools/dpdk-devbind.py` for managing devices.

Find ports for binding to DPDK by running the tool with `-s` option.

```
$ $RTE_SDK/usertools/dpdk-devbind.py --status

Network devices using DPDK-compatible driver
============================================
<none>
```

(continues on next page)

(continued from previous page)

```
Network devices using kernel driver
====================================
0000:29:00.0 '82571EB ... 10bc' if=enp41s0f0 drv=e1000e unused=
0000:29:00.1 '82571EB ... 10bc' if=enp41s0f1 drv=e1000e unused=
0000:2a:00.0 '82571EB ... 10bc' if=enp42s0f0 drv=e1000e unused=
0000:2a:00.1 '82571EB ... 10bc' if=enp42s0f1 drv=e1000e unused=

Other Network devices
=====================
<none>
....
```

You can find network ports are bound to kernel driver and not to DPDK. To bind a port to DPDK, run `dpdk-devbind.py` with specifying a driver and a device ID. Device ID is a PCI address of the device or more friendly style like `eth0` found by `ifconfig` or `ip` command..

```
# Bind a port with 2a:00.0 (PCI address)
./usertools/dpdk-devbind.py --bind=uio_pci_generic 2a:00.0

# or eth0
./usertools/dpdk-devbind.py --bind=uio_pci_generic eth0
```

After binding two ports, you can find it is under the DPDK driver and cannot find it by using `ifconfig` or `ip`.

```
$ $RTE_SDK/usertools/dpdk-devbind.py -s

Network devices using DPDK-compatible driver
============================================
0000:2a:00.0 '82571EB ... 10bc' drv=uio_pci_generic unused=vfio-pci
0000:2a:00.1 '82571EB ... 10bc' drv=uio_pci_generic unused=vfio-pci

Network devices using kernel driver
===================================
0000:29:00.0 '...' if=enp41s0f0 drv=e1000e unused=vfio-pci,uio_pci_generic
0000:29:00.1 '...' if=enp41s0f1 drv=e1000e unused=vfio-pci,uio_pci_generic

Other Network devices
=====================
<none>
....
```

3.2.6 Confirm DPDK is setup properly

For testing, you can confirm if you are ready to use DPDK by running DPDK's sample application. `l2fwd` is good example to confirm it before SPP because it is very similar to SPP's worker process for forwarding.

```
$ cd $RTE_SDK/examples/l2fwd
$ make
  CC main.o
  LD l2fwd
  INSTALL-APP l2fwd
  INSTALL-MAP l2fwd.map
```

In this case, run this application simply with just two options while DPDK has many kinds of

options.

- `-l`: core list
- `-p`: port mask

```
$ sudo ./build/app/l2fwd \
  -l 1-2 \
  -- -p 0x3
```

It must be separated with `--` to specify which option is for EAL or application. Refer to L2 Forwarding Sample Application for more details.

3.2.7 Build Documentation

This documentation is able to be built as HTML and PDF formats from make command. Before compiling the documentation, you need to install some of packages required to compile.

For HTML documentation, install sphinx and additional theme.

```
$ pip3 install sphinx \
  sphinx-rtd-theme
```

For PDF, inkscape and latex packages are required.

```
# Ubuntu
$ sudo apt install inkscape \
  texlive-latex-extra \
  texlive-latex-recommended
```

```
# CentOS
$ sudo yum install inkscape \
  texlive-latex
```

You might also need to install `latexmk` in addition to if you use Ubuntu 18.04 LTS.

```
$ sudo apt install latexmk
```

HTML documentation is compiled by running make with `doc-html`. This command launch sphinx for compiling HTML documents. Compiled HTML files are created in `docs/guides/_build/html/` and You can find the top page `index.html` in the directory.

```
$ make doc-html
```

PDF documentation is compiled with `doc-pdf` which runs latex for. Compiled PDF file is created as `docs/guides/_build/html/SoftPatchPanel.pdf`.

```
$ make doc-pdf
```

You can also compile both of HTML and PDF documentations with `doc` or `doc-all`.

```
$ make doc
# or
$ make doc-all
```

Note: For CentOS, compilation PDF document is not supported.

3.3 How to Use

As described in *Design*, SPP consists of primary process for managing resources, secondary processes for forwarding packet, and SPP controller to accept user commands and send it to SPP processes.

You should keep in mind the order of launching processes if you do it manually, or you can use startup script. This start script is for launching `spp-ctl`, `spp_primary` and SPP CLI.

Before starting, you should define environmental variable `SPP_FILE_PREFIX` for using the same prefix among SPP processes. `--file-prefix` is an EAL option for using a different shared data file prefix for a DPDK process.

```
$ export SPP_FILE_PREFIX=spp
```

This option is used for running several DPDK processes because it is not allowed different processes to have the same name of share data file, although each process of multi-process application should have the same prefix on the contrary. Even if you do not run several DPDK applications, you do not need to define actually. However, it is a good practice because SPP is used for connecting DPDK applications in actual usecases.

3.3.1 Quick Start

Run `bin/start.sh` with configuration file `bin/config.sh`. However, at the first time you run the script, it is failed because this configration file does not exist. It create the config from template `bin/sample/config.sh` and asks you to edit this file. All of options for launching the processes are defined in the configuration file.

Edit the config file before run `bin/start.sh` again. It is expected you have two physical ports on your server, but it is configurable. You can use virtual ports instead of physical. The number of ports is defined as `PRI_PORTMASK=0x03` as default. If you do not have physical ports and use two memif ports instead of physical, uncomment `PRI_MEMIF_VDEVS=(0 1)`. You can also use several types of port at once.

```
# spp_primary options
...
PRI_PORTMASK=0x03  # total num of ports of spp_primary.

# Vdevs of spp_primary
#PRI_MEMIF_VDEVS=(0 1)  # IDs of `net_memif`
#PRI_VHOST_VDEVS=(11 12)  # IDs of `eth_vhost`
...
```

After that, you can run the startup script again for launching processes.

```
# launch with default URL http://127.0.0.1:7777
$ bin/start.sh
Start spp-ctl
Start spp_primary
```

(continues on next page)

(continued from previous page)

```
Waiting for spp_primary is ready ..................... OK! (8.5[sec])
Welcome to the SPP CLI. Type `help` or `?` to list commands.

spp >
```

Check status of `spp_primary` because it takes several seconds to be ready. Confirm that the status is `running`.

```
spp > status
- spp-ctl:
  - address: 127.0.0.1:7777
- primary:
  - status: running
- secondary:
  - processes:
```

Now you are ready to launch secondary processes from `pri; launch` command, or another terminal. Here is an example for launching `spp_nfv` with options from `pri; launch`. Log file of this process is created as `log/spp_nfv1.log`.

```
spp > pri; launch nfv 1 -l 1,2 -m 512 --file-prefix spp -- -n 1 -s ...
```

This `launch` command supports TAB completion. Parameters for `spp_nfv` are completed after secondary ID `1`.

You might notice `--file-prefix spp` which should be the same value among primary and secondary processes. SPP CLI expects that this value can be referred as environmental variable `SPP_FILE_PREFIX`, and spp_primary is launched with the same `--file-prefix spp`. If you run SPP from `bin/start.sh`, you do no need to define the variable by yourself because it is defined in `bin/config.sh` so that spp_primary is launched with the prefix.

```
spp > pri; launch nfv 1

# Press TAB
spp > pri; launch nfv 1 -l 1,2 -m 512 --file-prefix spp -- -n 1 -s ...
```

It is same as following options launching from terminal.

```
$ sudo ./src/nfv/x86_64-native-linux-gcc/spp_nfv \
    -l 1,2 -n 4 -m 512 \
    --proc-type secondary \
    --file-prefix spp \
    -- \
    -n 1 \
    -s 127.0.0.1:6666
```

Parameters for completion are defined in SPP CLI, and you can find parameters with `config` command.

```
spp > config
- max_secondary: "16"   # The maximum number of secondary processes
- prompt: "spp > "  # Command prompt
- topo_size: "60%"  # Percentage or ratio of topo
- sec_mem: "-m 512" # Mem size
...
```

You can launch consequence secondary processes from CLI for your usage. If you just patch

two DPDK applications on host, it is enough to use one `spp_nfv`, or use `spp_vf` if you need to classify packets.

```
spp > pri; launch nfv 2 -l 1,3 -m 512 --file-prefix spp -- -n 2 -s ...
spp > pri; launch vf 3 -l 1,4,5,6 -m 512 --file-prefix spp -- -n 3 -s ...
...
```

If you launch processes by yourself, `spp_primary` must be launched before secondary processes. `spp-ctl` need to be launched before SPP CLI, but no need to be launched before other processes. SPP CLI is launched from `spp.py`. If `spp-ctl` is not running after primary and secondary processes are launched, processes wait `spp-ctl` is launched.

In general, `spp-ctl` should be launched first, then SPP CLI and `spp_primary` in each of terminals without running as background process. After `spp_primary`, you launch secondary processes for your usage.

In the rest of this chapter is for explaining how to launch each of processes options and usages for the all of processes. How to connect to VMs is also described in this chapter.

How to use of these secondary processes is described as usecases in the next chapter.

3.3.2 SPP Controller

SPP Controller consists of `spp-ctl` and SPP CLI.

spp-ctl

`spp-ctl` is a HTTP server for REST APIs for managing SPP processes. In default, it is accessed with URL `http://127.0.0.1:7777` or `http://localhost:7777`. `spp-ctl` shows no messages at first after launched, but shows log messages for events such as receiving a request or terminating a process.

```
# terminal 1
$ cd /path/to/spp
$ python3 src/spp-ctl/spp-ctl
```

It has a option `-b` for binding address explicitly to be accessed from other than default, `127.0.0.1` or `localhost`. If you deploy SPP on multiple nodes, you might need to use `-b` option it to be accessed from other processes running on other than local node.

```
# launch with URL http://192.168.1.100:7777
$ python3 src/spp-ctl/spp-ctl -b 192.168.1.100
```

`spp-ctl` is the most important process in SPP. For some usecases, you might better to manage this process with `systemd`. Here is a simple example of service file for systemd.

```
[Unit]
Description = SPP Controller

[Service]
ExecStart = /usr/bin/python3 /path/to/spp/src/spp-ctl/spp-ctl
User = root
```

All of options can be referred with help option `-h`.

```
$ python3 ./src/spp-ctl/spp-ctl -h
usage: spp-ctl [-h] [-b BIND_ADDR] [-p PRI_PORT]
               [-s SEC_PORT] [-a API_PORT]

SPP Controller

optional arguments:
  -h, --help            show this help message and exit
  -b BIND_ADDR, --bind-addr BIND_ADDR
                        bind address, default=localhost
  -p PRI_PORT           primary port, default=5555
  -s SEC_PORT           secondary port, default=6666
  -a API_PORT           web api port, default=7777
```

SPP CLI

If `spp-ctl` is launched, go to the next terminal and launch SPP CLI.

```
# terminal 2
$ cd /path/to/spp
$ python3 src/spp.py
Welcome to the spp.   Type help or ? to list commands.

spp >
```

If you launched `spp-ctl` with `-b` option, you also need to use the same option for `spp.py`, or failed to connect and to launch.

```
# terminal 2
# bind to spp-ctl on http://192.168.1.100:7777
$ python3 src/spp.py -b 192.168.1.100
Welcome to the spp.   Type help or ? to list commands.

spp >
```

One of the typical usecase of this option is to deploy multiple SPP nodes. Fig. 3.1 is an exmaple of multiple nodes case. There are three nodes on each of which `spp-ctl` is running for accepting requests for SPP. These `spp-ctl` processes are controlled from `spp.py` on host1 and all of paths are configured across the nodes. It is also able to be configured between hosts by changing soure or destination of phy ports.

Fig. 3.1: Multiple SPP nodes

Launch SPP CLI with three entries of binding addresses with `-b` option for specifying `spp-ctl`.

```
# Launch SPP CLI with three nodes
$ python3 src/spp.py -b 192.168.11.101 \
    -b 192.168.11.102 \
    -b 192.168.11.103 \
```

You can also add nodes after SPP CLI is launched.

```
# Launch SPP CLI with one node
$ python3 src/spp.py -b 192.168.11.101
Welcome to the SPP CLI. Type `help` or `?` to list commands.
```

(continues on next page)

(continued from previous page)

```
# Add the rest of nodes after
spp > server add 192.168.11.102
Registered spp-ctl "192.168.11.102:7777".
spp > server add 192.168.11.103
Registered spp-ctl "192.168.11.103:7777".
```

You find the host under the management of SPP CLI and switch with `server` command.

```
spp > server list
  1: 192.168.1.101:7777 *
  2: 192.168.1.102:7777
  3: 192.168.1.103:7777
```

To change the server, add an index number after `server`.

```
# Launch SPP CLI
spp > server 3
Switch spp-ctl to "3: 192.168.1.103:7777".
```

All of options can be referred with help option `-h`.

```
$ python3 src/spp.py -h
usage: spp.py [-h] [-b BIND_ADDR] [--wait-pri] [--config CONFIG]

SPP Controller

optional arguments:
  -h, --help            show this help message and exit
  -b BIND_ADDR, --bind-addr BIND_ADDR
                        bind address, default=127.0.0.1:7777
  --wait-pri            Wait for spp_primary is launched
  --config CONFIG       Config file path
```

All of SPP CLI commands are described in *SPP Commands*.

Default Configuration

SPP CLI imports several params from configuration file while launching. Some of behaviours of SPP CLI depends on the params. The default configuration is defined in `src/controller/config/default.yml`. You can change this params by editing the config file, or from `config` command after SPP CLI is launched.

All of config params are referred by `config` command.

```
# show list of config
spp > config
- max_secondary: "16"       # The maximum number of secondary processes
- sec_nfv_nof_lcores: "1"   # Default num of lcores for workers of spp_nfv
....
```

To change the config, set a value for the param. Here is an example for changing command prompt.

```
# set prompt to "$ spp "
spp > config prompt "$ spp "
```

(continues on next page)

(continued from previous page)

```
Set prompt: "$ spp "
$ spp
```

3.3.3 SPP Primary

SPP primary is a resource manager and has a responsibility for initializing EAL for secondary processes. It should be launched before secondary.

To launch SPP primary, run `spp_primary` with specific options.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 0 -n 4 \
    --socket-mem 512,512 \
    --huge-dir /dev/hugepages \
    --proc-type primary \
    --file-prefix $SPP_FILE_PREFIX \
    --base-virtaddr 0x100000000
    -- \
    -p 0x03 \
    -n 10 \
    -s 192.168.1.100:5555
```

SPP primary takes EAL options and application specific options.

Core list option `-l` is for assigining cores and SPP primary requires just one core. You can use core mask option `-c` instead of `-l`. For memory, this example is for reserving 512 MB on each of two NUMA nodes hardware, so you use `-m 1024` simply, or `--socket-mem 1024,0` if you run the process on single NUMA node.

Note: If you use DPDK v18.08 or before, you should consider give `--base-virtaddr` with 4 GiB or higher value because a secondary process is accidentally failed to mmap while init memory. The reason of the failure is secondary process tries to reserve the region which is already used by some of thread of primary.

```
# Failed to secondary
EAL: Could not mmap 17179869184 ... - please use '--base-virtaddr' option
```

`--base-virtaddr` is to decide base address explicitly to avoid this overlapping. 4 GiB `0x100000000` is enough for the purpose.

If you use DPDK v18.11 or later, `--base-virtaddr 0x100000000` is enabled in default. You need to use this option only for changing the default value.

If `spp_primary` is launched with two or more lcores, forwarder or monitor is activated. The default is forwarder and monitor is optional in this case. If you use monitor thread, additional option `--disp-stat` is required. Here is an example for launching `spp_primary` with monitor thread.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 0-1 -n 4 \    # two lcores
    --socket-mem 512,512 \
```

(continues on next page)

(continued from previous page)

```
    --huge-dir /dev/hugepages \
    --proc-type primary \
    --file-prefix $SPP_FILE_PREFIX \
    --base-virtaddr 0x100000000
    -- \
    -p 0x03 \
    -n 10 \
    -s 192.168.1.100:5555
    --disp-stats
```

Primary process sets up physical ports of given port mask with `-p` option and ring ports of the number of `-n` option. Ports of `-p` option is for accepting incomming packets and `-n` option is for inter-process packet forwarding. You can also add ports initialized with `--vdev` option to physical ports. However, ports added with `--vdev` cannot referred from secondary processes.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 0 -n 4 \
    --socket-mem 512,512 \
    --huge-dir=/dev/hugepages \
    --vdev eth_vhost1,iface=/tmp/sock1  # used as 1st phy port
    --vdev eth_vhost2,iface=/tmp/sock2  # used as 2nd phy port
    --proc-type=primary \
    --file-prefix $SPP_FILE_PREFIX \
    --base-virtaddr 0x100000000
    -- \
    -p 0x03 \
    -n 10 \
    -s 192.168.1.100:5555
```

In case of using MLX5 supported NIC, you must add `dv_flow_en=1` with white list option.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 0 -n 4 \
    -w 0000:03:00.0,dv_flow_en=1 \
    -w 0000:04:00.0,dv_flow_en=1 \
    -w 0000:05:00.0 \
    --socket-mem 512,512 \
    --huge-dir /dev/hugepages \
    --proc-type primary \
    --base-virtaddr 0x100000000
    -- \
    -p 0x03 \
    -n 10 \
    -s 192.168.1.100:5555
```

- EAL options:
  - `-l`: core list
  - `--socket-mem`: Memory size on each of NUMA nodes.
  - `--huge-dir`: Path of hugepage dir.
  - `--proc-type`: Process type.
  - `--base-virtaddr`: Specify base virtual address.
  - `--disp-stats`: Show statistics periodically.

- Application options:
  - `-p`: Port mask.
  - `-n`: Number of ring PMD.
  - `-s`: IP address of controller and port prepared for primary.

3.3.4 SPP Secondary

Secondary process behaves as a client of primary process and a worker for doing tasks for packet processing. There are several kinds of secondary process, for example, simply forwarding between ports, classsifying packets by referring its header or duplicate packets for redundancy.

spp_nfv

Run `spp_nfv` with options which simply forward packets as similar as `l2fwd`.

```
# terminal 4
$ cd /path/to/spp
$ sudo ./src/nfv/x86_64-native-linux-gcc/spp_nfv \
    -l 2-3 -n 4 \
    --proc-type secondary \
    --file-prefix $SPP_FILE_PREFIX \
    -- \
    -n 1 \
    -s 192.168.1.100:6666
```

EAL options are the same as primary process. Here is a list of application options of `spp_nfv`.

- `-n`: Secondary ID.
- `-s`: IP address and secondary port of spp-ctl.
- `--vhost-client`: Enable vhost-user client mode.

Secondary ID is used to identify for sending messages and must be unique among all of secondaries. If you attempt to launch a secondary process with the same ID, it is failed.

If `--vhost-client` option is specified, then `vhost-user` act as the client, otherwise the server. For reconnect feature from SPP to VM, `--vhost-client` option can be used. This reconnect features requires QEMU 2.7 (or later). See also Vhost Sample Application.

spp_vf

`spp_vf` is a kind of secondary process for classify or merge packets.

```
$ sudo ./src/vf/x86_64-native-linux-gcc/spp_vf \
    -l 2-13 -n 4 \
    --proc-type secondary \
    --file-prefix $SPP_FILE_PREFIX \
    -- \
    --client-id 1 \
    -s 192.168.1.100:6666 \
    --vhost-client
```

EAL options are the same as primary process. Here is a list of application options of `spp_vf`.

- `--client-id`: Client ID unique among secondary processes.
- `-s`: IPv4 address and secondary port of spp-ctl.
- `--vhost-client`: Enable vhost-user client mode.

spp_mirror

`spp_mirror` is a kind of secondary process for duplicating packets, and options are same as `spp_vf`.

```
$ sudo ./src/mirror/x86_64-native-linux-gcc/spp_mirror \
    -l 2,3 -n 4 \
    --proc-type secondary \
    --file-prefix $SPP_FILE_PREFIX \
    -- \
    --client-id 1 \
    -s 192.168.1.100:6666 \
    --vhost-client
```

EAL options are the same as primary process. Here is a list of application options of `spp_mirror`.

- `--client-id`: Client ID unique among secondary processes.
- `-s`: IPv4 address and secondary port of spp-ctl.
- `--vhost-client`: Enable vhost-user client mode.

spp_pcap

Other than PCAP feature implemented as pcap port in `spp_nfv`, SPP provides `spp_pcap` for capturing comparatively heavy traffic.

```
$ sudo ./src/pcap/x86_64-native-linux-gcc/spp_pcap \
    -l 2-5 -n 4 \
    --proc-type secondary \
    --file-prefix $SPP_FILE_PREFIX \
    -- \
    --client-id 1 \
    -s 192.168.1.100:6666 \
    -c phy:0 \
    --out-dir /path/to/dir \
    --fsize 107374182
```

EAL options are the same as primary process. Here is a list of application options of `spp_pcap`.

- `--client-id`: Client ID unique among secondary processes.
- `-s`: IPv4 address and secondary port of spp-ctl.
- `-c`: Captured port. Only `phy` and `ring` are supported.
- `--out-dir`: Optional. Path of dir for captured file. Default is `/tmp`.
- `--fsize`: Optional. Maximum size of a capture file. Default is `1GiB`.

Captured file of LZ4 is generated in `/tmp` by default. The name of file is consists of timestamp, resource ID of captured port, ID of `writer` threads and sequential number. Timestamp is decided when capturing is started and formatted as `YYYYMMDDhhmmss`. Both of `writer` thread ID and sequential number are started from `1`. Sequential number is required for the case if the size of captured file is reached to the maximum and another file is generated to continue capturing.

This is an example of captured file. It consists of timestamp, `20190214154925`, port `phy0`, thread ID `1` and sequential number `1`.

```
/tmp/spp_pcap.20190214154925.phy0.1.1.pcap.lz4
```

`spp_pcap` also generates temporary files which are owned by each of `writer` threads until capturing is finished or the size of captured file is reached to the maximum. This temporary file has additional extension `tmp` at the end of file name.

```
/tmp/spp_pcap.20190214154925.phy0.1.1.pcap.lz4.tmp
```

Launch from SPP CLI

You can launch SPP secondary processes from SPP CLI wihtout openning other terminals. `pri; launch` command is for any of secondary processes with specific options. It takes secondary type, ID and options of EAL and application itself as similar to launching from terminal. Here is an example of launching `spp_nfv`. You notice that there is no `--proc-type secondary` which should be required for secondary. It is added to the options by SPP CLI before launching the process.

```
# terminal 2
# launch spp_nfv with sec ID 2
spp > pri; launch nfv 2 -l 1,2 -m 512 -- -n 2 -s 192.168.1.100:6666
Send request to launch nfv:2.
```

After running this command, you can find `nfv:2` is launched successfully.

```
# terminal 2
spp > status
- spp-ctl:
  - address: 192.168.1.100:7777
- primary:
  - status: running
- secondary:
  - processes:
    1: nfv:2
```

Instead of displaying log messages in terminal, it outputs the messages in a log file. All of log files of secondary processes launched with `pri` are located in `log/` directory under the project root. The name of log file is found `log/spp_nfv-2.log`.

```
# terminal 5
$ tail -f log/spp_nfv-2.log
SPP_NFV: Used lcores: 1 2
SPP_NFV: entering main loop on lcore 2
SPP_NFV: My ID 2 start handling message
SPP_NFV: [Press Ctrl-C to quit ...]
SPP_NFV: Creating socket...
```

(continues on next page)

(continued from previous page)

```
SPP_NFV: Trying to connect ... socket 24
SPP_NFV: Connected
SPP_NFV: Received string: _get_client_id
SPP_NFV: token 0 = _get_client_id
SPP_NFV: To Server: {"results":[{"result":"success"}],"client_id":2, ...
```

Launch SPP on VM

To communicate DPDK application running on a VM, it is required to create a virtual device for the VM. In this instruction, launch a VM with qemu command and create `vhost-user` and `virtio-net-pci` devices on the VM.

Before launching VM, you need to prepare a socket file for creating `vhost-user` device. Run `add` command with resource UID `vhost:0` to create socket file.

```
# terminal 2
spp > nfv 1; add vhost:0
```

In this example, it creates socket file with index 0 from `spp_nfv` of ID 1. Socket file is created as `/tmp/sock0`. It is used as a qemu option to add vhost interface.

Launch VM with `qemu-system-x86_64` for x86 64bit architecture. Qemu takes many options for defining resources including virtual devices. You cannot use this example as it is because some options are depend on your environment. You should specify disk image with `-hda`, sixth option in this example, and `qemu-ifup` script for assigning an IP address for the VM to be able to access as 12th line.

```
# terminal 5
$ sudo qemu-system-x86_64 \
    -cpu host \
    -enable-kvm \
    -numa node,memdev=mem \
    -mem-prealloc \
    -hda /path/to/image.qcow2 \
    -m 4096 \
    -smp cores=4,threads=1,sockets=1 \
    -object \
    memory-backend-file,id=mem,size=4096M,mem-path=/dev/hugepages,share=on \
    -device e1000,netdev=net0,mac=00:AD:BE:B3:11:00 \
    -netdev tap,id=net0,ifname=net0,script=/path/to/qemu-ifup \
    -nographic \
    -chardev socket,id=chr0,path=/tmp/sock0 \  # /tmp/sock0
    -netdev vhost-user,id=net1,chardev=chr0,vhostforce \
    -device virtio-net-pci,netdev=net1,mac=00:AD:BE:B4:11:00 \
    -monitor telnet::44911,server,nowait
```

This VM has two network interfaces. `-device e1000` is a management network port which requires `qemu-ifup` to activate while launching. Management network port is used for login and setup the VM. `-device virtio-net-pci` is created for SPP or DPDK application running on the VM.

`vhost-user` is a backend of `virtio-net-pci` which requires a socket file `/tmp/sock0` created from secondary with `-chardev` option.

For other options, please refer to QEMU User Documentation.

Note: In general, you need to prepare several qemu images for launcing several VMs, but installing DPDK and SPP for several images is bother and time consuming.

You can shortcut this tasks by creating a template image and copy it to the VMs. It is just one time for installing for template.

After VM is booted, you install DPDK and SPP in the VM as in the host. IP address of the VM is assigned while it is created and you can find the address in a file generated from libvirt if you use Ubuntu.

```
# terminal 5
$ cat /var/lib/libvirt/dnsmasq/virbr0.status
[
    {
        "ip-address": "192.168.122.100",
        ...

# Login VM, install DPDK and SPP
$ ssh user@192.168.122.100
...
```

It is recommended to configure `/etc/default/grub` for hugepages and reboot the VM after installation.

Finally, login to the VM, bind ports to DPDK and launch `spp-ctl` and `spp_primamry`. You should add `-b` option to be accessed from SPP CLI on host.

```
# terminal 5
$ ssh user@192.168.122.100
$ cd /path/to/spp
$ python3 src/spp-ctl/spp-ctl -b 192.168.122.100
...
```

Confirm that virtio interfaces are under the management of DPDK before launching DPDK processes.

```
# terminal 6
$ ssh user@192.168.122.100
$ cd /path/to/spp
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 1 -n 4 \
    -m 1024 \
    --huge-dir=/dev/hugepages \
    --proc-type=primary \
    --base-virtaddr 0x100000000
    --file-prefix $SPP_FILE_PREFIX \
    -- \
    -p 0x03 \
    -n 6 \
    -s 192.168.122.100:5555
```

You can configure SPP running on the VM from SPP CLI. Use `server` command to switch node under the management.

```
# terminal 2
# show list of spp-ctl nodes
spp > server
```

(continues on next page)

(continued from previous page)

```
1: 192.168.1.100:7777 *
2: 192.168.122.100:7777

# change node under the management
spp > server 2
Switch spp-ctl to "2: 192.168.122.100:7777".

# confirm node is switched
spp > server
1: 192.168.1.100:7777
2: 192.168.122.100:7777 *

# configure SPP on VM
spp > status
...
```

Now, you are ready to setup your network environment for DPDK and non-DPDK applications with SPP. SPP enables users to configure service function chaining between applications running on host and VMs. Usecases of network configuration are explained in the next chapter.

Using virsh

First of all, please check version of qemu.

```
$ qemu-system-x86_64 --version
```

You should install qemu 2.7 or higher for using vhost-user client mode. Refer instruction to install.

`virsh` is a command line interface that can be used to create, destroy, stop start and edit VMs and configure.

You also need to install following packages to run `virt-install`.

- libvirt-bin
- virtinst
- bridge-utils

virt-install

Install OS image with `virt-install` command. `--location` is a URL of installer. Use Ubuntu 16.04 for amd64 in this case.

```
http://archive.ubuntu.com/ubuntu/dists/xenial/main/installer-amd64/
```

This is an example of `virt-install`.

```
virt-install \
--name VM_NAME \
--ram 4096 \
--disk path=/var/lib/libvirt/images/VM_NAME.img,size=30 \
--vcpus 4 \
--os-type linux \
```

(continues on next page)

(continued from previous page)

```
--os-variant ubuntu16.04 \
--network network=default \
--graphics none \
--console pty,target_type=serial \
--location 'http://archive.ubuntu.com/ubuntu/dists/xenial/main/...'
--extra-args 'console=ttyS0,115200n8 serial'
```

You might need to enable serial console as following.

```
$sudo systemctl enable serial-getty@ttyS0.service
$sudo systemctl start serial-getty@ttyS0.service
```

Edit Config

Edit configuration of VM with virsh command. The name of VMs are found from `virsh list`.

```
# Find the name of VM
$ sudo virsh list --all

$ sudo virsh edit VM_NAME
```

You need to define namespace `qemu` to use tags such as `<qemu:commandline>`. In `libvirt`, `<qemu:commandline>` tag is supported to utilize qemu specific features. In this example configuration of hugepage and/or network device is done via modifying domain XML file. Please see details in libvirt document.

```
xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'
```

In addition, you need to add attributes for specific resources for DPDK and SPP.

- `<memoryBacking>`
- `<qemu:commandline>`

Take care about the index numbers of devices should be the same value such as `chr0` or `sock0` in `virtio-net-pci` device. This index is referred as ID of vhost port from SPP. MAC address defined in the attribute is used while registering destinations for classifier's table.

```
<qemu:arg value='virtio-net-pci,netdev=vhost-net0,mac=52:54:00:12:34:56'/>
```

Here is an example of XML config for using with SPP. The following example is just excerpt from complete sample. The complete sample can be found in spp-vm1.xml.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>spp-vm1</name>
  <uuid>d90f5420-861a-4479-8559-62d7a1545cb9</uuid>
  <memory unit='KiB'>4194304</memory>
  <currentMemory unit='KiB'>4194304</currentMemory>
  "..."
  <qemu:commandline>
    <qemu:arg value='-cpu'/>
    <qemu:arg value='host'/>
    <qemu:arg value='-object'/>
    <qemu:arg value='memory-backend-file,
    id=mem,size=4096M,mem-path=/run/hugepages/kvm,share=on'/>
    <qemu:arg value='-numa'/>
```

(continues on next page)

(continued from previous page)

```
    <qemu:arg value='node,memdev=mem'/>
    <qemu:arg value='-mem-prealloc'/>
    <qemu:arg value='-chardev'/>
    <qemu:arg value='socket,id=chr0,path=/tmp/sock0,server'/>
    <qemu:arg value='-device'/>
    <qemu:arg value='virtio-net-pci,netdev=vhost-net0,
    mac=52:54:00:12:34:56'/>
    <qemu:arg value='-netdev'/>
    <qemu:arg value='vhost-user,id=vhost-net0,chardev=chr0,vhostforce'/>
    <qemu:arg value='-chardev'/>
    <qemu:arg value='socket,id=chr1,path=/tmp/sock1,server'/>
    <qemu:arg value='-device'/>
    <qemu:arg value='virtio-net-pci,netdev=vhost-net1,
    mac=52:54:00:12:34:57'/>
    <qemu:arg value='-netdev'/>
    <qemu:arg value='vhost-user,id=vhost-net1,chardev=chr1,vhostforce'/>
  </qemu:commandline>
</domain>
```

Launch VM

After updating XML configuration, launch VM with `virsh start`.

```
$ virsh start VM_NAME
```

It is required to add network configurations for processes running on the VMs. If this configuration is skipped, processes cannot communicate with others via SPP.

On the VMs, add an interface and disable offload.

```
# Add interface
$ sudo ifconfig IF_NAME inet IPADDR netmask NETMASK up

# Disable offload
$ sudo ethtool -K IF_NAME tx off
```

3.4 Performance Optimization

3.4.1 Reduce Context Switches

Use the `isolcpus` Linux kernel parameter to isolate them from Linux scheduler to reduce context switches. It prevents workloads of other processes than DPDK running on reserved cores with `isolcpus` parameter.

For Ubuntu 16.04, define `isolcpus` in `/etc/default/grub`.

```
GRUB_CMDLINE_LINUX_DEFAULT="isolcpus=0-3,5,7"
```

The value of this `isolcpus` depends on your environment and usage. This example reserves six cores(0,1,2,3,5,7).

3.4.2 Optimizing QEMU Performance

QEMU process runs threads for vcpu emulation. It is effective strategy for pinning vcpu threads to decicated cores.

To find vcpu threads, you use `ps` command to find PID of QEMU process and `pstree` command for threads launched from QEMU process.

```
$ ps ea
   PID TTY      STAT   TIME COMMAND
192606 pts/11   Sl+    4:42 ./x86_64-softmmu/qemu-system-x86_64 -cpu host ...
```

Run `pstree` with `-p` and this PID to find all threads launched from QEMU.

```
$ pstree -p 192606
qemu-system-x86(192606)--+--{qemu-system-x8}(192607)
                         |--{qemu-system-x8}(192623)
                         |--{qemu-system-x8}(192624)
                         |--{qemu-system-x8}(192625)
                         |--{qemu-system-x8}(192626)
```

Update affinity by using `taskset` command to pin vcpu threads. The vcpu threads is listed from the second entry and later. In this example, assign PID 192623 to core 4, PID 192624 to core 5 and so on.

```
$ sudo taskset -pc 4 192623
pid 192623's current affinity list: 0-31
pid 192623's new affinity list: 4
$ sudo taskset -pc 5 192624
pid 192624's current affinity list: 0-31
pid 192624's new affinity list: 5
$ sudo taskset -pc 6 192625
pid 192625's current affinity list: 0-31
pid 192625's new affinity list: 6
$ sudo taskset -pc 7 192626
pid 192626's current affinity list: 0-31
pid 192626's new affinity list: 7
```

3.4.3 Consideration of NUMA node

`spp_primary` tries to create memory pool in the same NUMA node where it is launched. Under NUMA configuration, the NUMA node where `spp_primary` is launched and the NUMA node where NIC is connected can be different (e.g. spp_primary runs in NUMA node 0 while NIC is connected with NUMA node 1). Such configuration may cause performance degradation. In general, under NUMA configuration, it is best practice to use CPU and NIC which belongs to the same NUMA node for best performance. So user should align those when performance degradation makes the situation critical.

To check NUMA node which CPU/NIC core belongs, `lstopo` command can be used. In the following example, CPU core 0 belongs to NUMA node 0 while `enp175s0f0` belongs to NUMA node 1.

```
$ lstopo
Machine (93GB total)
  NUMANode L#0 (P#0 46GB)
    Package L#0 + L3 L#0 (17MB)
```

(continues on next page)

(continued from previous page)

```
      L2 L#0 (1024KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0
.....
  NUMANode L#1 (P#1 47GB)
    Package L#1 + L3 L#1 (17MB)
      L2 L#12 (1024KB) + L1d L#12 (32KB) + L1i L#12 (32KB) + Core L#12
        PU L#24 (P#1)
        PU L#25 (P#25)
.....
    HostBridge L#10
      PCIBridge
        PCI 8086:1563
          Net L#10 "enp175s0f0"
        PCI 8086:1563
          Net L#11 "enp175s0f1"
```

CPU core where `spp_primary` run can be specified using -l option.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 0 -n 4 \
    --socket-mem 512,512 \
    --huge-dir /dev/hugepages \
    --proc-type primary \
    --file-prefix $SPP_FILE_PREFIX \
    --base-virtaddr 0x100000000
    -- \
    -p 0x03 \
    -n 10 \
    -s 192.168.1.100:5555
```

3.4.4 Reference

- [1] Best pinning strategy for latency/performance trade-off
- [2] PVP reference benchmark setup using testpmd
- [3] Enabling Additional Functionality
- [4] How to get best performance with NICs on Intel platforms

CHAPTER 4

Use Cases

As described in *Design*, SPP has several kinds of secondary process for usecases such as simple forwarding to network entities, capturing or mirroring packets for monitoring, or connecting VMs or containers for Service Function Chaining in NFV.

This chapter is focusing on explaining about each of secondary processes with simple usecases. Usecase of `spp_primary` is not covered here because it is almost similar to `spp_nfv` and no need to explain both.

Details of usages of each process is not covered in this chapter. You can refer the details of SPP processes via CLI from *SPP Commands*, or via REST API from *API Reference*.

4.1 spp_nfv

4.1.1 Single spp_nfv

The most simple usecase mainly for testing performance of packet forwarding on host. One `spp_nfv` and two physical ports.

In this usecase, try to configure two senarios.

- Configure `spp_nfv` as L2fwd
- Configure `spp_nfv` for Loopback

First of all, Check the status of `spp_nfv` from SPP CLI.

```
spp > nfv 1; status
- status: idling
- lcore_ids:
  - master: 1
  - slave: 2
- ports:
  - phy:0
  - phy:1
```

This status message explains that `nfv 1` has two physical ports.

Configure spp_nfv as L2fwd

Assing the destination of ports with `patch` subcommand and start forwarding. Patch from `phy:0` to `phy:1` and `phy:1` to `phy:0`, which means it is bi-directional connection.

```
spp > nfv 1; patch phy:0 phy:1
Patch ports (phy:0 -> phy:1).
spp > nfv 1; patch phy:1 phy:0
Patch ports (phy:1 -> phy:0).
spp > nfv 1; forward
Start forwarding.
```

Confirm that status of `nfv 1` is updated to `running` and ports are patches as you defined.

```
spp > nfv 1; status
- status: running
- lcore_ids:
  - master: 1
  - slave: 2
- ports:
  - phy:0 -> phy:1
  - phy:1 -> phy:0
```

Fig. 4.1: spp_nfv as l2fwd

Stop forwarding and reset patch to clear configuration. `patch reset` is to clear all of patch configurations.

```
spp > nfv 1; stop
Stop forwarding.
spp > nfv 1; patch reset
Clear all of patches.
```

Configure spp_nfv for Loopback

Patch `phy:0` to `phy:0` and `phy:1` to `phy:1` for loopback.

```
spp > nfv 1; patch phy:0 phy:0
Patch ports (phy:0 -> phy:0).
spp > nfv 1; patch phy:1 phy:1
Patch ports (phy:1 -> phy:1).
spp > nfv 1; forward
Start forwarding.
```

4.1.2 Dual spp_nfv

Use case for testing performance of packet forwarding with two `spp_nfv` on host. Throughput is expected to be better than *Single spp_nfv* usecase because bi-directional forwarding of single `spp_nfv` is shared with two of uni-directional forwarding between dual `spp_nfv`.

In this usecase, configure two senarios almost similar to previous section.

- Configure Two `spp_nfv` as L2fwd
- Configure Two `spp_nfv` for Loopback

Configure Two spp_nfv as L2fwd

Assing the destination of ports with `patch` subcommand and start forwarding. Patch from `phy:0` to `phy:1` for `nfv 1` and from `phy:1` to `phy:0` for `nfv 2`.

```
spp > nfv 1; patch phy:0 phy:1
Patch ports (phy:0 -> phy:1).
spp > nfv 2; patch phy:1 phy:0
Patch ports (phy:1 -> phy:0).
spp > nfv 1; forward
Start forwarding.
spp > nfv 2; forward
Start forwarding.
```

Fig. 4.2: Two spp_nfv as l2fwd

Configure two spp_nfv for Loopback

Patch `phy:0` to `phy:0` for `nfv 1` and `phy:1` to `phy:1` for `nfv 2` for loopback.

```
spp > nfv 1; patch phy:0 phy:0
Patch ports (phy:0 -> phy:0).
spp > nfv 2; patch phy:1 phy:1
Patch ports (phy:1 -> phy:1).
spp > nfv 1; forward
Start forwarding.
spp > nfv 2; forward
Start forwarding.
```

Fig. 4.3: Two spp_nfv for loopback

4.1.3 Dual spp_nfv with Ring PMD

In this usecase, configure two senarios by using ring PMD.

- Uni-Directional L2fwd
- Bi-Directional L2fwd

Ring PMD

Ring PMD is an interface for communicating between secondaries on host. The maximum number of ring PMDs is defined as `-n` option of `spp_primary` and ring ID is started from 0.

Ring PMD is added by using `add` subcommand. All of ring PMDs is showed with `status` subcommand.

```
spp > nfv 1; add ring:0
Add ring:0.
spp > nfv 1; status
- status: idling
- lcore_ids:
  - master: 1
  - slave: 2
- ports:
  - phy:0
  - phy:1
  - ring:0
```

Notice that `ring:0` is added to `nfv 1`. You can delete it with `del` command if you do not need to use it anymore.

```
spp > nfv 1; del ring:0
Delete ring:0.
spp > nfv 1; status
- status: idling
- lcore_ids:
  - master: 1
  - slave: 2
- ports:
  - phy:0
  - phy:1
```

Uni-Directional L2fwd

Add a ring PMD and connect two `spp_nvf` processes. To configure network path, add `ring:0` to `nfv 1` and `nfv 2`. Then, connect it with `patch` subcommand.

```
spp > nfv 1; add ring:0
Add ring:0.
spp > nfv 2; add ring:0
Add ring:0.
spp > nfv 1; patch phy:0 ring:0
Patch ports (phy:0 -> ring:0).
spp > nfv 2; patch ring:0 phy:1
Patch ports (ring:0 -> phy:1).
spp > nfv 1; forward
Start forwarding.
spp > nfv 2; forward
Start forwarding.
```

Fig. 4.4: Uni-Directional l2fwd

Bi-Directional L2fwd

Add two ring PMDs to two `spp_nvf` processes. For bi-directional forwarding, patch `ring:0` for a path from `nfv 1` to `nfv 2` and `ring:1` for another path from `nfv 2` to `nfv 1`.

First, add `ring:0` and `ring:1` to `nfv 1`.

```
spp > nfv 1; add ring:0
Add ring:0.
spp > nfv 1; add ring:1
Add ring:1.
spp > nfv 1; status
- status: idling
- lcore_ids:
  - master: 1
  - slave: 2
- ports:
  - phy:0
  - phy:1
  - ring:0
  - ring:1
```

Then, add `ring:0` and `ring:1` to `nfv 2`.

```
spp > nfv 2; add ring:0
Add ring:0.
spp > nfv 2; add ring:1
Add ring:1.
spp > nfv 2; status
- status: idling
- lcore_ids:
  - master: 1
  - slave: 3
- ports:
  - phy:0
  - phy:1
  - ring:0
  - ring:1
```

```
spp > nfv 1; patch phy:0 ring:0
Patch ports (phy:0 -> ring:0).
spp > nfv 1; patch ring:1 phy:0
Patch ports (ring:1 -> phy:0).
spp > nfv 2; patch phy:1 ring:1
Patch ports (phy:1 -> ring:0).
spp > nfv 2; patch ring:0 phy:1
Patch ports (ring:0 -> phy:1).
spp > nfv 1; forward
Start forwarding.
spp > nfv 2; forward
Start forwarding.
```

Fig. 4.5: Bi-Directional l2fwd

4.1.4 Single spp_nfv with Vhost PMD

Vhost PMD

Vhost PMD is an interface for communicating between on hsot and guest VM. As described in *How to Use*, vhost must be created by `add` subcommand before the VM is launched.

Setup Vhost PMD

In this usecase, add `vhost:0` to `nfv 1` for communicating with the VM. First, check if `/tmp/sock0` is already exist. You should remove it already exist to avoid a failure of socket file creation.

```
# remove sock0 if already exist
$ ls /tmp | grep sock
sock0 ...
$ sudo rm /tmp/sock0
```

Create `/tmp/sock0` from `nfv 1`.

```
spp > nfv 1; add vhost:0
Add vhost:0.
```

Setup Network Configuration in spp_nfv

Launch a VM by using the vhost interface created in the previous step. Lauunching VM is described in *How to Use*.

Patch `phy:0` to `vhost:0` and `vhost:1` to `phy:1` from `nfv 1` running on host.

```
spp > nfv 1; patch phy:0 vhost:0
Patch ports (phy:0 -> vhost:0).
spp > nfv 1; patch vhost:1 phy:1
Patch ports (vhost:1 -> phy:1).
spp > nfv 1; forward
Start forwarding.
```

Finally, start forwarding inside the VM by using two vhost ports to confirm that network on host is configured.

```
$ sudo $RTE_SDK/examples/build/l2fwd -l 0-1 -- -p 0x03
```

Fig. 4.6: Single spp_nfv with vhost PMD

4.1.5 Single spp_nfv with PCAP PMD

PCAP PMD

Pcap PMD is an interface for capturing or restoring traffic. For usign pcap PMD, you should set `CONFIG_RTE_LIBRTE_PMD_PCAP` and `CONFIG_RTE_PORT_PCAP` to `y` and compile DPDK before SPP. Refer to *Install DPDK and SPP* for details of setting up.

Pcap PMD has two different streams for rx and tx. Tx device is for capturing packets and rx is for restoring captured packets. For rx device, you can use any of pcap files other than SPP's pcap PMD.

To start using pcap pmd, just using `add` subcommand as ring. Here is an example for creating pcap PMD `pcap:1`.

```
spp > nfv 1; add pcap:1
```

After running it, you can find two of pcap files in `/tmp`.

```
$ ls /tmp | grep pcap$
spp-rx1.pcap
spp-tx1.pcap
```

If you already have a dumped file, you can use it by it putting as `/tmp/spp-rx1.pcap` before running the `add` subcommand. SPP does not overwrite rx pcap file if it already exist, and it just overwrites tx pcap file.

Capture Incoming Packets

As the first usecase, add a pcap PMD and capture incoming packets from `phy:0`.

```
spp > nfv 1; add pcap 1
Add pcap:1.
spp > nfv 1; patch phy:0 pcap:1
Patch ports (phy:0 -> pcap:1).
spp > nfv 1; forward
Start forwarding.
```

Fig. 4.7: Rapture incoming packets

In this example, we use pktgen. Once you start forwarding packets from pktgen, you can see that the size of `/tmp/spp-tx1.pcap` is increased rapidly (or gradually, it depends on the rate).

```
Pktgen:/> set 0 size 1024
Pktgen:/> start 0
```

To stop capturing, simply stop forwarding of `spp_nfv`.

```
spp > nfv 1; stop
Stop forwarding.
```

You can analyze the dumped pcap file with other tools like as wireshark.

Restore dumped Packets

In this usecase, use dumped file in previsou section. Copy `spp-tx1.pcap` to `spp-rx2.pcap` first.

```
$ sudo cp /tmp/spp-tx1.pcap /tmp/spp-rx2.pcap
```

Then, add pcap PMD `pcap:2` to another `spp_nfv`.

```
spp > nfv 2; add pcap:2
Add pcap:2.
```

You can find that `spp-tx2.pcap` is creaeted and `spp-rx2.pcap` still remained.

Fig. 4.8: Restore dumped packets

```
$ ls -al /tmp/spp*.pcap
-rw-r--r-- 1 root root         24  ...  /tmp/spp-rx1.pcap
-rw-r--r-- 1 root root 2936703640  ...  /tmp/spp-rx2.pcap
-rw-r--r-- 1 root root 2936703640  ...  /tmp/spp-tx1.pcap
-rw-r--r-- 1 root root          0  ...  /tmp/spp-tx2.pcap
```

To confirm packets are restored, patch `pcap:2` to `phy:1` and watch received packets on pktgen.

```
spp > nfv 2; patch pcap:2 phy:1
Patch ports (pcap:2 -> phy:1).
spp > nfv 2; forward
Start forwarding.
```

After started forwarding, you can see that packet count is increased.

4.2 spp_vf

`spp_vf` is a secondary process for providing L2 classification as a simple pusedo SR-IOV features.

Note: `--file-prefix` option is not required in this section because there is not DPDK application other than SPP.

4.2.1 Classify ICMP Packets

To confirm classifying packets, sends ICMP packet from remote node by using ping and watch the response. Incoming packets through `NIC0` are classified based on destination address.

Fig. 4.9: Network Configuration

Setup

Launch `spp-ctl` and SPP CLI before primary and secondary processes.

```
# terminal 1
$ python3 ./src/spp-ctl/spp-ctl -b 192.168.1.100
```

```
# terminal 2
$ python3 ./src/spp.py -b 192.168.1.100
```

`spp_primary` on the second lcore with `-l 0` and two ports `-p 0x03`.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 1 -n 4 \
    --socket-mem 512,512 \
    --huge-dir=/run/hugepages/kvm \
    --proc-type=primary \
    -- \
    -p 0x03 \
    -n 10 -s 192.168.1.100:5555
```

After `spp_primary` is launched, run secondary process `spp_vf`. In this case, lcore options is `-l 2-6` for one master thread and four worker threads.

```
# terminal 4
$ sudo ./src/vf/x86_64-native-linux-gcc/spp_vf \
   -l 2-6 \
   -n 4 --proc-type=secondary \
   -- \
   --client-id 1 \
   -s 192.168.1.100:6666 \
```

Network Configuration

Configure network as described in Fig. 4.9 step by step.

First of all, setup worker threads from `component` command with lcore ID and other options on local host `host2`.

```
# terminal 2
spp > vf 1; component start cls 3 classifier
spp > vf 1; component start fwd1 4 forward
spp > vf 1; component start fwd2 5 forward
spp > vf 1; component start mgr 6 merge
```

Add ports for each of components as following. The number of rx and tx ports are different for each of component's role.

```
# terminal 2

# classifier
spp > vf 1; port add phy:0 rx cls
spp > vf 1; port add ring:0 tx cls
spp > vf 1; port add ring:1 tx cls

# forwarders
spp > vf 1; port add ring:0 rx fwd1
spp > vf 1; port add ring:2 tx fwd1
spp > vf 1; port add ring:1 rx fwd2
spp > vf 1; port add ring:3 tx fwd2

# merger
spp > vf 1; port add ring:2 rx mgr
spp > vf 1; port add ring:3 rx mgr
spp > vf 1; port add phy:1 tx mgr
```

You also need to configure MAC address table for classifier. In this case, you need to register two MAC addresses. Although any MAC can be used, you use `52:54:00:12:34:56` and `52:54:00:12:34:58`.

```
# terminal 2
spp > vf 1; classifier_table add mac 52:54:00:12:34:56 ring:0
spp > vf 1; classifier_table add mac 52:54:00:12:34:58 ring:1
```

Send Packet from Remote Host

Ensure NICs, `ens0` and `ens1` in this case, are upped on remote host `host1`. You can up by using ifconfig if the status is down.

```
# terminal 1 on remote host
# Configure ip address of ens0
$ sudo ifconfig ens0 192.168.140.1 netmask 255.255.255.0 up
```

Add arp entries of MAC addresses statically to be resolved.

```
# terminal 1 on remote host
# set MAC address
$ sudo arp -i ens0 -s 192.168.140.2 52:54:00:12:34:56
$ sudo arp -i ens0 -s 192.168.140.3 52:54:00:12:34:58
```

Start tcpdump command for capturing `ens1`.

```
# terminal 2 on remote host
$ sudo tcpdump -i ens1
```

Then, start ping in other terminals.

```
# terminal 3 on remote host
# ping via NIC0
$ ping 192.168.140.2
```

```
# terminal 4 on remote host
# ping via NIC0
$ ping 192.168.140.3
```

You can see ICMP Echo requests are received from ping on terminal 2.

Shutdown spp_vf Components

Basically, you can shutdown all of SPP processes with `bye all` command. This section describes graceful shutting down. First, delete entries of `classifier_table` and ports of components.

```
# terminal 2
# Delete MAC address from Classifier
spp > vf 1; classifier_table del mac 52:54:00:12:34:56 ring:0
spp > vf 1; classifier_table del mac 52:54:00:12:34:58 ring:1
```

```
# terminal 2
# classifier
spp > vf 1; port del phy:0 rx cls
spp > vf 1; port del ring:0 tx cls
spp > vf 1; port del ring:1 tx cls
```

(continues on next page)

(continued from previous page)

```
# forwarders
spp > vf 1; port del ring:0 rx fwd1
spp > vf 1; port del ring:2 tx fwd1
spp > vf 1; port del ring:1 rx fwd2
spp > vf 1; port del ring:3 tx fwd2

# mergers
spp > vf 1; port del ring:2 rx mgr
spp > vf 1; port del ring:3 rx mgr
spp > vf 1; port del phy:1 tx mgr
```

Then, stop components.

```
# terminal 2
spp > vf 1; component stop cls
spp > vf 1; component stop fwd1
spp > vf 1; component stop fwd2
spp > vf 1; component stop mgr
```

You can confirm that worker threads are cleaned from `status`.

```
spp > vf 1; status
Basic Information:
  - client-id: 1
  - ports: [phy:0, phy:1]
  - lcore_ids:
    - master: 2
    - slaves: [3, 4, 5, 6]
Classifier Table:
  No entries.
Components:
  - core:3 '' (type: unuse)
  - core:4 '' (type: unuse)
  - core:5 '' (type: unuse)
  - core:6 '' (type: unuse)
```

Finally, terminate `spp_vf` by using `exit` or `bye sec`.

```
spp > vf 1; exit
```

4.2.2 SSH Login to VMs

This usecase is to classify packets for ssh connections as another example. Incoming packets are classified based on destination addresses and reterned packets are aggregated before going out.

Fig. 4.10: Simple SSH Login

Setup

Launch `spp-ctl` and SPP CLI before primary and secondary processes.

```
# terminal 1
$ python3 ./src/spp-ctl/spp-ctl -b 192.168.1.100
```

```
# terminal 2
$ python3 ./src/spp.py -b 192.168.1.100
```

`spp_primary` on the second lcore with `-l 1` and two ports `-p 0x03`.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 1 -n 4 \
    --socket-mem 512,512 \
    --huge-dir=/run/hugepages/kvm \
    --proc-type=primary \
    -- \
    -p 0x03 -n 10 -s 192.168.1.100:5555
```

Then, run secondary process `spp_vf` with `-l 0,2-13` which indicates to use twelve lcores.

```
# terminal 4
$ sudo ./src/vf/x86_64-native-linux-gcc/spp_vf \
    -l 0,2-13 \
    -n 4 --proc-type=secondary \
    -- \
    --client-id 1 \
    -s 192.168.1.100:6666 --vhost-client
```

Network Configuration

Detailed netowrk configuration of Fig. 4.10 is described below. In this usecase, use two NICs on each of host1 and host2 for redundancy.

Incoming packets through NIC0 or NIC1 are classified based on destionation address.

Fig. 4.11: Network Configuration SSH with spp_vhost

You need to input a little bit large amount of commands for the configuration, or use `playback` command to load from config files. You can load network configuration from recipes in `recipes/usecases/` as following.

```
# terminal 2
# Load config from recipe
spp > playback recipes/usecases/spp_vf/ssh/1-start_components.rcp
spp > playback recipes/usecases/spp_vf/ssh/2-add_port_path1.rcp
....
```

First of all, start components with names such as `cls1`, `fwd1` or so.

```
# terminal 2
spp > vf 1; component start cls1 2 classifier
spp > vf 1; component start fwd1 3 forward
spp > vf 1; component start fwd2 4 forward
spp > vf 1; component start fwd3 5 forward
spp > vf 1; component start fwd4 6 forward
spp > vf 1; component start mgr1 7 merge
```

Each of components must have rx and tx ports for forwarding. Add ports for each of components as following. You notice that classifier has two tx ports and merger has two rx ports.

```
# terminal 2
# classifier
spp > vf 1; port add phy:0 rx cls1
spp > vf 1; port add ring:0 tx cls1
spp > vf 1; port add ring:1 tx cls1

# forwarders
spp > vf 1; port add ring:0 rx fwd1
spp > vf 1; port add vhost:0 tx fwd1
spp > vf 1; port add ring:1 rx fwd2
spp > vf 1; port add vhost:2 tx fwd2
spp > vf 1; port add vhost:0 rx fwd3
spp > vf 1; port add ring:2 tx fwd3
spp > vf 1; port add vhost:2 rx fwd4
spp > vf 1; port add ring:3 tx fwd4

# merger
spp > vf 1; port add ring:2 rx mgr1
spp > vf 1; port add ring:3 rx mgr1
spp > vf 1; port add phy:0 tx mgr1
```

Classifier component decides the destination with MAC address by referring `classifier_table`. MAC address and corresponging port is registered to the table. In this usecase, you need to register two MAC addresses of targetting VM for mgr1, and also mgr2 later.

```
# terminal 2
# Register MAC addresses for mgr1
spp > vf 1; classifier_table add mac 52:54:00:12:34:56 ring:0
spp > vf 1; classifier_table add mac 52:54:00:12:34:58 ring:1
```

Configuration for the second login path is almost the same as the first path.

```
# terminal 2
spp > vf 1; component start cls2 8 classifier
spp > vf 1; component start fwd5 9 forward
spp > vf 1; component start fwd6 10 forward
spp > vf 1; component start fwd7 11 forward
spp > vf 1; component start fwd8 12 forward
spp > vf 1; component start mgr2 13 merge
```

Add ports to each of components.

```
# terminal 2
# classifier
spp > vf 1; port add phy:1 rx cls2
spp > vf 1; port add ring:4 tx cls2
spp > vf 1; port add ring:5 tx cls2

# forwarders
spp > vf 1; port add ring:4 rx fwd5
spp > vf 1; port add vhost:1 tx fwd5
spp > vf 1; port add ring:5 rx fwd6
spp > vf 1; port add vhost:3 tx fwd6
spp > vf 1; port add vhost:1 rx fwd7
spp > vf 1; port add ring:6 tx fwd7
spp > vf 1; port add vhost:3 rx fwd8
spp > vf 1; port add ring:7 tx fwd8

# merger
```

(continues on next page)

(continued from previous page)

```
spp > vf 1; port add ring:6 rx mgr2
spp > vf 1; port add ring:7 rx mgr2
spp > vf 1; port add phy:1 tx mgr2
```

Register MAC address entries to `classifier_table` for `cls2`.

```
# terminal 2
# Register MAC address to classifier
spp > vf 1; classifier_table add mac 52:54:00:12:34:57 ring:4
spp > vf 1; classifier_table add mac 52:54:00:12:34:59 ring:5
```

Setup VMs

Launch two VMs with virsh command. Setup for virsh is described in *Using virsh*. In this case, VMs are named as `spp-vm1` and `spp-vm2`.

```
# terminal 5
$ virsh start spp-vm1   # VM1
$ virsh start spp-vm2   # VM2
```

After VMs are launched, login to `spp-vm1` first to configure.

Note: To avoid asked for unknown keys while login VMs, use `-o StrictHostKeyChecking=no` option for ssh.

```
$ ssh -o StrictHostKeyChecking=no sppuser at 192.168.122.31
```

Up interfaces and disable TCP offload to avoid ssh login is failed.

```
# terminal 5
# up interfaces
$ sudo ifconfig ens4 inet 192.168.140.21 netmask 255.255.255.0 up
$ sudo ifconfig ens5 inet 192.168.150.22 netmask 255.255.255.0 up

# disable TCP offload
$ sudo ethtool -K ens4 tx off
$ sudo ethtool -K ens5 tx off
```

Configuration of `spp-vm2` is almost similar to `spp-vm1`.

```
# terminal 5
# up interfaces
$ sudo ifconfig ens4 inet 192.168.140.31 netmask 255.255.255.0 up
$ sudo ifconfig ens5 inet 192.168.150.32 netmask 255.255.255.0 up

# disable TCP offload
$ sudo ethtool -K ens4 tx off
$ sudo ethtool -K ens5 tx off
```

Login to VMs

Now, you can login to VMs from the remote host1.

```
# terminal 5
# spp-vm1 via NIC0
$ ssh sppuser@192.168.140.21

# spp-vm1 via NIC1
$ ssh sppuser@192.168.150.22

# spp-vm2 via NIC0
$ ssh sppuser@192.168.140.31

# spp-vm2 via NIC1
$ ssh sppuser@192.168.150.32
```

Shutdown spp_vf Components

Basically, you can shutdown all of SPP processes with `bye all` command. This section describes graceful shutting down.

First, delete entries of `classifier_table` and ports of components for the first SSH login path.

```
# terminal 2
# Delete MAC address from table
spp > vf 1; classifier_table del mac 52:54:00:12:34:56 ring:0
spp > vf 1; classifier_table del mac 52:54:00:12:34:58 ring:1
```

Delete ports.

```
# terminal 2
# classifier
spp > vf 1; port del phy:0 rx cls1
spp > vf 1; port del ring:0 tx cls1
spp > vf 1; port del ring:1 tx cls1

# forwarders
spp > vf 1; port del ring:0 rx fwd1
spp > vf 1; port del vhost:0 tx fwd1
spp > vf 1; port del ring:1 rx fwd2
spp > vf 1; port del vhost:2 tx fwd2
spp > vf 1; port del vhost:0 rx fwd3
spp > vf 1; port del ring:2 tx fwd3
spp > vf 1; port del vhost:2 rx fwd4
spp > vf 1; port del ring:3 tx fwd4

# merger
spp > vf 1; port del ring:2 rx mgr1
spp > vf 1; port del ring:3 rx mgr1
spp > vf 1; port del phy:0 tx mgr1
```

Then, stop components.

```
# terminal 2
# Stop component to spp_vf
spp > vf 1; component stop cls1
spp > vf 1; component stop fwd1
spp > vf 1; component stop fwd2
spp > vf 1; component stop fwd3
spp > vf 1; component stop fwd4
spp > vf 1; component stop mgr1
```

Second, do termination for the second path. Delete entries from the table and ports from each of components.

```
# terminal 2
# Delete MAC address from Classifier
spp > vf 1; classifier_table del mac 52:54:00:12:34:57 ring:4
spp > vf 1; classifier_table del mac 52:54:00:12:34:59 ring:5
```

```
# terminal 2
# classifier2
spp > vf 1; port del phy:1 rx cls2
spp > vf 1; port del ring:4 tx cls2
spp > vf 1; port del ring:5 tx cls2

# forwarder
spp > vf 1; port del ring:4 rx fwd5
spp > vf 1; port del vhost:1 tx fwd5
spp > vf 1; port del ring:5 rx fwd6
spp > vf 1; port del vhost:3 tx fwd6
spp > vf 1; port del vhost:1 rx fwd7
spp > vf 1; port del ring:6 tx fwd7
spp > vf 1; port del vhost:3 rx fwd8
spp > vf 1; port del ring:7 tx fwd8

# merger
spp > vf 1; port del ring:6 rx mgr2
spp > vf 1; port del ring:7 rx mgr2
spp > vf 1; port del phy:1 tx mgr2
```

Then, stop components.

```
# terminal 2
# Stop component to spp_vf
spp > vf 1; component stop cls2
spp > vf 1; component stop fwd5
spp > vf 1; component stop fwd6
spp > vf 1; component stop fwd7
spp > vf 1; component stop fwd8
spp > vf 1; component stop mgr2
```

Exit spp_vf

Terminate spp_vf.

```
# terminal 2
spp > vf 1; exit
```

4.3 spp_mirror

Note: `--file-prefix` option is not required in this section because there is not DPDK application other than SPP.

4.3.1 Duplicate Packets

Simply duplicate incoming packets and send to two destinations. Remote `host1` sends ARP packets by using ping command and `spp_mirror` running on local `host2` duplicates packets to destination ports.

Network Configuration

Detailed configuration is described in Fig. 4.12. In this diagram, incoming packets from `phy:0` are mirrored. In `spp_mirror` process, worker thread `mir` copies incoming packets and sends to two destinations `phy:1` and `phy:2`.

Fig. 4.12: Duplicate packets with spp_mirror

Setup SPP

Change directory to spp and confirm that it is already compiled.

```
$ cd /path/to/spp
```

Launch `spp-ctl` before launching SPP primary and secondary processes. You also need to launch `spp.py` if you use `spp_mirror` from CLI. `-b` option is for binding IP address to communicate other SPP processes, but no need to give it explicitly if `127.0.0.1` or `localhost` .

```
# terminal 1
# Launch spp-ctl
$ python3 ./src/spp-ctl/spp-ctl -b 192.168.1.100
```

```
# terminal 2
# Launch SPP CLI
$ python3 ./src/spp.py -b 192.168.1.100
```

Start `spp_primary` with core list option `-l 1` and three ports `-p 0x07`.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 1 -n 4 \
    --socket-mem 512,512 \
    --huge-dir=/run/hugepages/kvm \
    --proc-type=primary \
    -- \
    -p 0x07 -n 10 -s 192.168.1.100:5555
```

Launch spp_mirror

Run secondary process `spp_mirror`.

```
# terminal 4
$ sudo ./src/mirror/x86_64-native-linux-gcc/app/spp_mirror \
 -l 0,2 -n 4 \
```

(continues on next page)

(continued from previous page)

```
 --proc-type secondary \
 -- \
 --client-id 1 \
 -s 192.168.1.100:6666 \
```

Start mirror component with core ID 2.

```
# terminal 2
spp > mirror 1; component start mir 2 mirror
```

Add `phy:0` as rx port, and `phy:1` and `phy:2` as tx ports.

```
# terminal 2
# add ports to mir
spp > mirror 1; port add phy:0 rx mir
spp > mirror 1; port add phy:1 tx mir
spp > mirror 1; port add phy:2 tx mir
```

Duplicate Packets

To check packets are mirrored, you run tcpdump for `ens1` and `ens2`. As you run ping for `ens0` next, you will see the same ARP requests trying to resolve `192.168.140.21` on terminal 1 and 2.

```
# terminal 1 at host1
# capture on ens1
$ sudo tcpdump -i ens1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens1, link-type EN10MB (Ethernet), capture size 262144 bytes
21:18:44.183261 ARP, Request who-has 192.168.140.21 tell R740n15, length 28
21:18:45.202182 ARP, Request who-has 192.168.140.21 tell R740n15, length 28
....
```

```
# terminal 2 at host1
# capture on ens2
$ sudo tcpdump -i ens2
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ens2, link-type EN10MB (Ethernet), capture size 262144 bytes
21:18:44.183261 ARP, Request who-has 192.168.140.21 tell R740n15, length 28
21:18:45.202182 ARP, Request who-has 192.168.140.21 tell R740n15, length 28
...
```

Start to send ARP request with ping.

```
# terminal 3 at host1
# send packet from NIC0
$ ping 192.168.140.21 -I ens0
```

Stop Mirroring

Delete ports for components.

```
# Delete port for mir
spp > mirror 1; port del phy:0 rx mir
```

(continues on next page)

(continued from previous page)

```
spp > mirror 1; port del phy:1 tx mir
spp > mirror 1; port del phy:2 tx mir
```

Next, stop components.

```
# Stop mirror
spp > mirror 1; component stop mir 2 mirror

spp > mirror 1; status
Basic Information:
  - client-id: 1
  - ports: [phy:0, phy:1]
  - lcore_ids:
    - master: 0
    - slave: 2
Components:
  - core:2 '' (type: unuse)
```

Finally, terminate `spp_mirror` to finish this usecase.

```
spp > mirror 1; exit
```

4.3.2 Monitoring Packets

Duplicate classified packets for monitoring before going to a VM. In this usecase, we are only interested in packets going to `VM1`. Although you might be able to run packet monitor app on host, run monitor on `VM3` considering more NFV like senario. You use `spp_mirror` for copying, and `spp_vf` classifying packets.

Fig. 4.13: Monitoring with spp_mirror

Setup SPP and VMs

Launch `spp-ctl` before launching SPP primary and secondary processes. You also need to launch `spp.py` if you use `spp_vf` from CLI. `-b` option is for binding IP address to communicate other SPP processes, but no need to give it explicitly if `127.0.0.1` or `localhost` although doing explicitly in this example to be more understandable.

```
# terminal 1
$ python3 ./src/spp-ctl/spp-ctl -b 192.168.1.100
```

```
# terminal 2
$ python3 ./src/spp.py -b 192.168.1.100
```

Start spp_primary with core list option `-l 1`.

```
# terminal 3
# Type the following in different terminal
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 1 -n 4 \
    --socket-mem 512,512 \
    --huge-dir=/run/hugepages/kvm \
```

(continues on next page)

(continued from previous page)

```
    --proc-type=primary \
    -- \
    -p 0x03 \
    -n 10 -s 192.168.1.100:5555
```

Netowrk Configuration

Detailed configuration of Fig. 4.13 is described in Fig. 4.14. In this senario, worker thread `mir` copies incoming packets from though `ring:0`. Then, sends to orignal destination `VM1` and anohter one `VM3`.

Fig. 4.14: Network configuration of monitoring packets

Launch `VM1`, `VM2` and `spp_vf` with core list `-l 0,2-8`.

```
# terminal 4
$ sudo ./src/vf/x86_64-native-linux-gcc/spp_vf \
    -l 0,2-8 \
    -n 4 --proc-type secondary \
    -- \
    --client-id 1 \
    -s 192.168.1.100:6666 \
    --vhost-client
```

Start components in `spp_vf`.

```
# terminal 2
spp > vf 1; component start cls 2 classifier
spp > vf 1; component start mgr 3 merge
spp > vf 1; component start fwd1 4 forward
spp > vf 1; component start fwd2 5 forward
spp > vf 1; component start fwd3 6 forward
spp > vf 1; component start fwd4 7 forward
spp > vf 1; component start fwd5 8 forward
```

Add ports for components.

```
# terminal 2
spp > vf 1; port add phy:0 rx cls
spp > vf 1; port add ring:0 tx cls
spp > vf 1; port add ring:1 tx cls

spp > vf 1; port add ring:2 rx mgr
spp > vf 1; port add ring:3 rx mgr
spp > vf 1; port add phy:0 tx mgr

spp > vf 1; port add ring:5 rx fwd1
spp > vf 1; port add vhost:0 tx fwd1

spp > vf 1; port add ring:1 rx fwd2
spp > vf 1; port add vhost:2 tx fwd2

spp > vf 1; port add vhost:1 rx fwd3
spp > vf 1; port add ring:2 tx fwd3

spp > vf 1; port add vhost:3 rx fwd4
```

(continues on next page)

(continued from previous page)

```
spp > vf 1; port add ring:3 tx fwd4

spp > vf 1; port add ring:4 rx fwd5
spp > vf 1; port add vhost:4 tx fwd5
```

Add classifier table entries.

```
# terminal 2
spp > vf 1; classifier_table add mac 52:54:00:12:34:56 ring:0
spp > vf 1; classifier_table add mac 52:54:00:12:34:58 ring:1
```

Launch spp_mirror

Run `spp_mirror`.

```
# terminal 6
$ sudo ./src/mirror/x86_64-native-linux-gcc/app/spp_mirror \
  -l 0,9 \
  -n 4 --proc-type secondary \
  -- \
  --client-id 2 \
  -s 192.168.1.100:6666 \
  --vhost-client
```

Start mirror component with lcore ID 9.

```
# terminal 2
spp > mirror 2; component start mir 9 mirror
```

Add `ring:0` as rx port, `ring:4` and `ring:5` as tx ports.

```
# terminal 2
spp > mirror 2; port add ring:0 rx mir
spp > mirror 2; port add ring:4 tx mir
spp > mirror 2; port add ring:5 tx mir
```

Receive Packet on VM3

You can capture incoming packets on `VM3` and compare it with on `VM1`. To capture incoming packets , use tcpdump for the interface, `ens4` in this case.

```
# terminal 5
# capture on ens4 of VM1
$ tcpdump -i ens4
```

```
# terminal 7
# capture on ens4 of VM3
$ tcpdump -i ens4
```

You send packets from the remote `host1` and confirm packets are received. IP address is the same as *Usecase of spp_vf*.

```
# Send packets from host1
$ ping 192.168.140.21
```

Stop Mirroring

Graceful shutdown of secondary processes is same as previous usecases.

4.4 spp_pcap

Note: `--file-prefix` option is not required in this section because there is not DPDK application other than SPP.

4.4.1 Packet Capture

This section describes a usecase for capturing packets with `spp_pcap`. See inside of the captured file with `tcpdump` command. Fig. 4.15 shows the overview of scenario in which incoming packets via `phy:0` are dumped as compressed pcap files by using `spp_pcap`.

Fig. 4.15: Packet capture with spp_pcap

Launch spp_pcap

Change directory if you are not in SPP's directory, and compile if not done yet.

```
$ cd /path/to/spp
```

Launch spp-ctl and SPP CLI in different terminals.

```
# terminal 1
$ python3 ./src/spp-ctl/spp-ctl -b 192.168.1.100
```

```
# terminal 2
$ python3 ./src/spp.py -b 192.168.1.100
```

Then, run `spp_primary` with one physical port.

```
# terminal 3
$ sudo ./src/primary/x86_64-native-linux-gcc/spp_primary \
    -l 0 -n 4 \
    --socket-mem 512,512 \
    --huge-dir /run/hugepages/kvm \
    --proc-type primary \
    -- \
    -p 0x01 \
    -n 8 -s 192.168.1.100:5555
```

After `spp_primary` is launched successfully, run `spp_pcap` in other terminal. In this usecase, you use default values for optional arguments. Output directory of captured file is `/tmp` and the size of file is `1GiB`. You notice that six lcores are assigned with `-l 1-6`. It means that you use one locre for master, one for receiver, and four for writer threads.

```
# terminal 4
$ sudo ./src/pcap/x86_64-native-linux-gcc/spp_pcap \
   -l 1-6 -n 4 --proc-type=secondary \
   -- \
   --client-id 1 -s 192.168.1.100:6666 \
   -c phy:0
```

You can confirm lcores and worker threads running on from `status` command.

```
# terminal 2
spp > pcap 1; status
Basic Information:
  - client-id: 1
  - status: idle
  - lcore_ids:
    - master: 1
    - slaves: [2, 3, 4, 5, 6]
Components:
  - core:2 receive
    - rx: phy:0
  - core:3 write
    - filename:
  - core:4 write
    - filename:
  - core:5 write
    - filename:
  - core:6 write
    - filename:
```

Start Capture

If you already started to send packets to `phy:0` from outside, you are ready to start capturing packets.

```
# terminal 2
spp > pcap 1; start
Start packet capture.
```

As you run `start` command, PCAP files are generated for each of `writer` threads for capturing.

```
# terminal 2
spp > pcap 1; status
Basic Information:
  - client-id: 1
  - status: running
  - lcore_ids:
    - master: 1
    - slaves: [2, 3, 4, 5, 6]
Components:
  - core:2 receive
    - rx: phy:0
  - core:3 write
    - filename: /tmp/spp_pcap.20190214161550.phy0.1.1.pcap.lz4
  - core:4 write
    - filename: /tmp/spp_pcap.20190214161550.phy0.2.1.pcap.lz4
  - core:5 write
    - filename: /tmp/spp_pcap.20190214161550.phy0.3.1.pcap.lz4
```

(continues on next page)

(continued from previous page)

```
  - core:6 write
    - filename: /tmp/spp_pcap.20190214161550.phy0.4.1.pcap.lz4
```

Stop Capture

Stop capturing and confirm that compressed PCAP files are generated.

```
# terminal 2
spp > pcap 1; stop
spp > ls /tmp
....
spp_pcap.20190214175446.phy0.1.1.pcap.lz4
spp_pcap.20190214175446.phy0.1.2.pcap.lz4
spp_pcap.20190214175446.phy0.1.3.pcap.lz4
spp_pcap.20190214175446.phy0.2.1.pcap.lz4
spp_pcap.20190214175446.phy0.2.2.pcap.lz4
spp_pcap.20190214175446.phy0.2.3.pcap.lz4
....
```

Index in the filename, such as `1.1` or `1.2`, is a combination of `writer` thread ID and sequenceal number. In this case, it means each of four threads generate three files.

Shutdown spp_pcap

Run `exit` or `bye sec` command to terminate `spp_pcap`.

```
# terminal 2
spp > pcap 1; exit
```

Inspect PCAP Files

You can inspect captured PCAP files by using utilities.

Merge PCAP Files

Extract and merge compressed PCAP files.

For extract several LZ4 files at once, use `-d` and `-m` options. `-d` is for decompression and `-m` is for multiple files.

You had better not to merge divided files into single file, but still several files because the size of merged file might be huge. Each of extracted PCAP file is 1GiB in default, so total size of extracted files is 12GiB in this case. To avoid the situation, merge files for each of threads and generate four PCAP files of 3GiB.

First, extract LZ4 files of writer thread ID 1.

```
# terminal 4
$ lz4 -d -m /tmp/spp_pcap.20190214175446.phy0.1.*
```

And confirm that the files are extracted.

```
# terminal 4
$ ls /tmp | grep pcap$
spp_pcap.20190214175446.phy0.1.1.pcap
spp_pcap.20190214175446.phy0.1.2.pcap
spp_pcap.20190214175446.phy0.1.3.pcap
```

Run `mergecap` command to merge extracted files to current directory as `spp_pcap1.pcap`.

```
# terminal 4
$ mergecap /tmp/spp_pcap.20190214175446.phy0.1.*.pcap -w spp_pcap1.pcap
```

Inspect PCAP file

You can use any of applications, for instance `wireshark` or `tcpdump`, for inspecting PCAP file. To inspect the merged PCAP file, read packet data from `tcpdump` command in this usecase. `-r` option is to dump packet data in human readable format.

```
# terminal 4
$ tcpdump -r spp_pcap1.pcap | less
17:54:52.559783 IP 192.168.0.100.1234 > 192.168.1.1.5678: Flags [.], ...
17:54:52.559784 IP 192.168.0.100.1234 > 192.168.1.1.5678: Flags [.], ...
17:54:52.559785 IP 192.168.0.100.1234 > 192.168.1.1.5678: Flags [.], ...
17:54:52.559785 IP 192.168.0.100.1234 > 192.168.1.1.5678: Flags [.], ...
```

4.5 Multiple Nodes

SPP provides multi-node support for configuring network across several nodes from SPP CLI. You can configure each of nodes step by step.

In Fig. 4.16, there are four nodes on which SPP and service VMs are running. Host1 behaves as a patch panel for connecting between other nodes. A request is sent from a VM on host2 to a VM on host3 or host4. Host4 is a backup server for host3 and replaced with host3 by changing network configuration. Blue lines are paths for host3 and red lines are for host4, and changed alternatively.

Fig. 4.16: Multiple nodes example

4.5.1 Launch SPP on Multiple Nodes

Before SPP CLI, launch spp-ctl on each of nodes. You should give IP address with `-b` option to be accessed from outside of the node. This is an example for launching spp-ctl on host1.

```
# Launch on host1
$ python3 src/spp-ctl/spp-ctl -b 192.168.11.101
```

You also need to launch it on host2, host3 and host4 in each of terminals.

After all of spp-ctls are lauched, launch SPP CLI with four `-b` options for each of hosts. SPP CLI is able to be launched on any of nodes.

```
# Launch SPP CLI
$ python3 src/spp.py -b 192.168.11.101 \
    -b 192.168.11.102 \
    -b 192.168.11.103 \
    -b 192.168.11.104 \
```

Or you can add nodes after launching SPP CLI. Here is an example of launching it with first node, and adding the rest of nodes after.

```
# Launch SPP CLI
$ python3 src/spp.py -b 192.168.11.101
Welcome to the spp.  Type help or ? to list commands.

spp > server add 192.168.11.102
Registered spp-ctl "192.168.11.102:7777".
spp > server add 192.168.11.103
Registered spp-ctl "192.168.11.103:7777".
spp > server add 192.168.11.104
Registered spp-ctl "192.168.11.104:7777".
```

If you have succeeded to launch all of `spp-ctl` processes before, you can find them by running `sever list` command.

```
# Launch SPP CLI
spp > server list
  1: 192.168.1.101:7777 *
  2: 192.168.1.102:7777
  3: 192.168.1.103:7777
  4: 192.168.1.104:7777
```

You might notice that first entry is marked with `*`. It means that the current node under the management is the first node.

4.5.2 Switch Nodes

SPP CLI manages a node marked with `*`. If you configure other nodes, change the managed node with `server` command. Here is an example to switch to third node.

```
# Launch SPP CLI
spp > server 3
Switch spp-ctl to "3: 192.168.1.103:7777".
```

And the result after changed to host3.

```
spp > server list
  1: 192.168.1.101:7777
  2: 192.168.1.102:7777
  3: 192.168.1.103:7777 *
  4: 192.168.1.104:7777
```

You can also confirm this change by checking IP address of spp-ctl from `status` command.

```
spp > status
- spp-ctl:
  - address: 192.168.1.103:7777
- primary:
  - status: not running
...
```

4.5.3 Configure Patch Panel Node

First of all of the network configuration, setup blue lines on host1 described in Fig. 4.16. You should confirm the managed server is host1.

```
spp > server list
  1: 192.168.1.101:7777 *
  2: 192.168.1.102:7777
  ...
```

Patch two sets of physical ports and start forwarding.

```
spp > nfv 1; patch phy:1 phy:2
Patch ports (phy:1 -> phy:2).
spp > nfv 1; patch phy:3 phy:0
Patch ports (phy:3 -> phy:0).
spp > nfv 1; forward
Start forwarding.
```

4.5.4 Configure Service VM Nodes

It is almost similar as *Setup Network Configuration in spp_nfv* to setup for host2, host3, and host4.

For host2, swith server to host2 and run nfv commands.

```
# switch to server 2
spp > server 2
Switch spp-ctl to "2: 192.168.1.102:7777".

# configure
spp > nfv 1; add vhost:0
Add vhost:0.
spp > nfv 1; patch phy:0 vhost:0
Patch ports (phy:0 -> vhost:0).
spp > nfv 1; patch vhost:0 phy:1
Patch ports (vhost:0 -> phy:1).
spp > nfv 1; forward
Start forwarding.
```

Then, swith to host3 and host4 for doing the same configuration.

4.5.5 Change Path to Backup Node

Finally, change path from blue lines to red lines.

```
# switch to server 1
spp > server 1
Switch spp-ctl to "1: 192.168.1.101:7777".

# remove blue path
spp > nfv 1; stop
Stop forwarding.
spp > nfv 1; patch reset

# configure red path
```

(continues on next page)

(continued from previous page)

```
spp > nfv 2; patch phy:1 phy:4
Patch ports (phy:1 -> phy:4).
spp > nfv 2; patch phy:5 phy:0
Patch ports (phy:5 -> phy:0).
spp > nfv 2; forward
Start forwarding.
```

4.6 Hardware Offload

SPP provides hardware offload functions.

Note: We tested following use cases at Connect-X 5 by Mellanox only. Even if you cannot use these use cases on different NIC, we don't support.

4.6.1 Hardware Classification

Some hardware provides packet classification function based on L2 mac address. This use case shows you how to use L2 classification.

Setup

Before using hardware packet classification, you must setup number of queues in hardware.

In `bin/config.sh`.

```
PRI_PORT_QUEUE=(
 "0 rxq 10 txq 10"
 "1 rxq 16 txq 16"
)
```

Above example includes the line `0 rxq 10 txq 10`. `0` of this line specifies physical port number, `rxq 10` is for 10 rx-queues, `txq 10` is for 10 tx-queues.

You should uncomment the following block in `bin/config.sh` to indicate hardware white list. The option `dv_flow_en=1` is for MLX5 poll mode driver.

```
PRI_WHITE_LIST=(
 "0000:04:00.0,dv_flow_en=1"
 "0000:05:00.0"
)
```

After editing `bin/config.sh`, you can launch SPP as following.

```
$ bin/start.sh
Start spp-ctl
Start spp_primary
Waiting for spp_primary is ready ..................... OK! (8.5[sec])
Welcome to the SPP CLI. Type `help` or `?` to list commands.
spp >
```

Then, you can launch `spp_vf` like this.

```
spp > pri; launch vf 1 -l 2,3,4,5 -m 512 --file-prefix spp \
-- --client-id 1 -s 127.0.0.1:6666
...
```

Configuration

Before configure the flow of classifying packets, you can validate such rules by using `flow validate` command.

```
spp > pri; flow validate phy:0 ingress pattern eth dst is \
10:22:33:44:55:66 / end actions queue index 1 / end
spp > pri; flow validate phy:0 ingress pattern eth dst is \
10:22:33:44:55:67 / end actions queue index 2 / end
```

Then, you can configure flow using `flow create` command like this.

```
spp > pri; flow create phy:0 ingress pattern eth dst is \
10:22:33:44:55:66 / end actions queue index 1 / end
spp > pri; flow create phy:0 ingress pattern eth dst is \
10:22:33:44:55:67 / end actions queue index 2 / end
```

You can confirm created flows by using `flow list` or `flow status` commands. `flow list` command provides the flow information of specified physical port.

```
spp > pri; flow list phy:0
ID      Group   Prio    Attr    Rule
0       0       0       i--     ETH => QUEUE
1       0       0       i--     ETH => QUEUE
```

To get detailed information for specific rule. The following example shows the case where showing detailed information for rule ID `0` of `phy:0`.

```
spp > pri; flow status phy:0 0
Attribute:
  Group   Priority Ingress Egress Transfer
  0       0        true    false  false
Patterns:
  - eth:
    - spec:
      - dst: 10:22:33:44:55:66
      - src: 00:00:00:00:00:00
      - type: 0x0000
    - last:
    - mask:
      - dst: FF:FF:FF:FF:FF:FF
      - src: 00:00:00:00:00:00
      - type: 0x0000
Actions:
    - queue:
      - index: 1
spp >
```

In this use case, two components `fwd1` and `fwd2` simply forward the packet to multi-tx queues. You can start these components like this.

```
spp > vf 1; component start fwd1 2 forward
spp > vf 1; component start fwd2 3 forward
```

For each `fwd1` and `fwd2`, configure the rx port like this.

```
spp > vf 1; port add phy:0 nq 1 rx fwd1
spp > vf 1; port add phy:0 nq 2 rx fwd2
```

Then, you can configure tx ports like this.

```
spp > vf 1; port add phy:1 nq 1 tx fwd1
spp > vf 1; port add phy:1 nq 2 tx fwd2
```

For confirming above configuration, you can use ping and tcpdump as described in *Classify ICMP Packets*.

Also, when you destroy the flow created above, commands will be like the following.

```
spp > pri; flow destroy phy:0 0
spp > pri; flow destroy phy:0 1
```

Or you can destroy all rules on specific hardware by using `flow destroy` command with `ALL` parameter.

```
spp > pri; flow destroy phy:0 ALL
```

4.6.2 Manipulate VLAN tag

Some hardware provides VLAN tag manipulation function. This use case shows you the case where incoming VLAN tagged packet detagged and non-tagged packet tagged when outgoing using hardware offload function.

After having done above use case, you can continue to following. In this use case, we are assuming incoming packets which includes `vid=100` to `phy:0`, these vid will be removed(detagged) and transferred to `fwd1`. Tx packets from `fwd1` are sent to queue#0 on phy:1 with tagged by `vid=100`. Packets which includes `vid=200` to `phy:0` are to be sent to `fwd2` with removing the vid, Tx packets from `fwd2` are sent to `queue#1` on `phy:1` with tagged by `vid=200`.

For detagging flow creation.

```
spp > pri; flow create phy:0 ingress group 1 pattern eth dst is \
10:22:33:44:55:66 / vlan vid is 100 / end actions queue index 1 \
/ of_pop_vlan / end
spp > pri; flow create phy:0 ingress group 1 pattern eth dst is \
10:22:33:44:55:67 / vlan vid is 200 / end actions queue index 2 \
/ of_pop_vlan / end
spp > pri; flow create phy:0 ingress group 0 pattern eth / end \
actions jump group 1 / end
```

For tagging flow creation.

```
spp > pri; flow create phy:1 egress group 1 pattern eth dst is \
10:22:33:44:55:66 / end actions of_push_vlan ethertype 0x8100 \
/ of_set_vlan_vid vlan_vid 100 / of_set_vlan_pcp vlan_pcp 3 / end
spp > pri; flow create phy:1 egress group 1 pattern eth dst is \
10:22:33:44:55:67 / end actions of_push_vlan ethertype 0x8100 \
/ of_set_vlan_vid vlan_vid 200 / of_set_vlan_pcp vlan_pcp 3 / end
spp > pri; flow create phy:1 egress group 0 pattern eth / end \
actions jump group 1 / end
```

If you want to send vlan-tagged packets, the NIC connected to `phy:0` will be configured by following.

```
$ sudo ip l add link ens0 name ens0.100 type vlan id 100
$ sudo ip l add link ens0 name ens0.200 type vlan id 200
$ sudo ip a add 192.168.140.1/24 dev ens0.100
$ sudo ip a add 192.168.150.1/24 dev ens0.100
$ sudo ip l set ens0.100 up
$ sudo ip l set ens0.200 up
```

4.6.3 Connecting with VMs

This use case shows you how to configure hardware offload and VMs.

First, we should clean up flows and delete ports.

```
spp > vf 1; port del phy:0 nq 0 rx fwd1
spp > vf 1; port del phy:0 nq 1 rx fwd2
spp > vf 1; port del phy:1 nq 0 tx fwd1
spp > vf 1; port del phy:1 nq 1 tx fwd2
spp > pri; flow destroy phy:0 ALL
spp > pri; flow destroy phy:1 ALL
```

Configure flows.

```
spp > pri; flow create phy:0 ingress group 1 pattern eth dst is \
10:22:33:44:55:66 / vlan vid is 100 / end actions queue index 1 \
/ of_pop_vlan / end
spp > pri; flow create phy:0 ingress group 1 pattern eth dst is \
10:22:33:44:55:67 / vlan vid is 200 / end actions queue index 2 \
/ of_pop_vlan / end
spp > pri; flow create phy:0 ingress group 0 pattern eth / end \
actions jump group 1 / end
spp > pri; flow create phy:0 egress group 1 pattern eth src is \
10:22:33:44:55:66 / end actions of_push_vlan ethertype 0x8100 \
/ of_set_vlan_vid vlan_vid 100 / of_set_vlan_pcp vlan_pcp 3 / end
spp > pri; flow create phy:0 egress group 1 pattern eth src is \
10:22:33:44:55:67 / end actions of_push_vlan ethertype 0x8100 \
/ of_set_vlan_vid vlan_vid 200 / of_set_vlan_pcp vlan_pcp 3 / end
spp > pri; flow create phy:0 egress group 0 pattern eth / end \
actions jump group 1 / end
```

Start components.

```
spp > vf 1; component start fwd3 4 forward
spp > vf 1; component start fwd4 5 forward
```

Start and setup two VMs as described in *SSH Login to VMs*. Add ports to forwarders.

```
spp > vf 1; port add phy:0 nq 1 rx fwd1
spp > vf 1; port add vhost:0 tx fwd1
spp > vf 1; port add phy:0 nq 2 rx fwd2
spp > vf 1; port add vhost:1 tx fwd2
spp > vf 1; port add vhost:0 rx fwd3
spp > vf 1; port add phy:0 nq 3 tx fwd3
spp > vf 1; port add vhost:1 rx fwd4
spp > vf 1; port add phy:0 nq 4 tx fwd4
```

Then you can login to each VMs.

Note that you must add arp entries of MAC addresses statically to be resolved.

```
# terminal 1 on remote host
# set MAC address
$ sudo arp -i ens0 -s 192.168.140.31 10:22:33:44:55:66
$ sudo arp -i ens0 -s 192.168.150.32 10:22:33:44:55:67
```

4.6.4 Reference

The following features are tested.

MT27710 Family [ConnectX-4 Lx] 1015 - dstMAC - dstMAC(range)

MT27800 Family [ConnectX-5] 1017 - dstMAC - dstMAC(range) - vlan vid - vlan vid+dstMAC - tagging+detagging

Ethernet Controller XXV710 for 25GbE SFP28 158b - dstMAC

4.7 Pipe PMD

Pipe PMD constitutes a virtual Ethernet device (named spp_pipe) using rings which the spp_primary allocated.

It is necessary for the DPDK application using spp_pipe to implement it as the secondary process under the spp_primary as the primary process.

Using spp_pipe enables high-speed packet transfer through rings among DPDK applications using spp_pipe and SPP secondary processes such as spp_nfv and spp_vf.

4.7.1 Using pipe PMD

Create a pipe port by requesting to the spp_primary to use spp_pipe beforehand. There are *CLI* and *REST API* to create a pipe port. A ring used for rx transfer and a ring used for tx transfer are specified at a pipe port creation.

For example creating `pipe:0` with `ring:0` for rx and `ring:1` for tx by CLI as follows.

```
spp > pri; add pipe:0 ring:0 ring:1
```

The name as the Ethernet device of `pipe:N` is `spp_pipeN`. DPDK application which is the secondary process of the spp_primary can get the port id of the device using `rte_eth_dev_get_port_by_name`.

Requirement of DPDK application using spp_pipe

It is necessary to use the common mbuf mempool of the SPP processes.

```
#define PKTMBUF_POOL_NAME "Mproc_pktmbuf_pool"

struct rte_mempool *mbuf_pool;

mbuf_pool = rte_mempool_lookup(PKTBBUF_POOL_NAME);
```

4.7.2 Use cases

Here are some examples using spp_pipe.

Note: A ring allocated by the spp_primary assumes it is single producer and single consumer. It is user responsibility that each ring in the model has single producer and single consumer.

Direct communication between applications

To create pipe ports by CLI before running applications as follows.

```
spp > pri; add pipe:0 ring:0 ring:1
spp > pri; add pipe:1 ring:1 ring:0
```

Fixed application chain using spp_nfv

To construct the model by CLI before running applications as follows.

```
spp > pri; add pipe:0 ring:0 ring:1
spp > pri; add pipe:1 ring:1 ring:2
spp > nfv 1; add ring:0
spp > nfv 1; patch phy:0 ring:0
spp > nfv 1; forward
spp > nfv 2; add ring:2
spp > nfv 2; patch ring:2 phy:1
spp > nfv 2; forward
```

Service function chaining using spp_vf

To construct the model by CLI before running applications as follows.

```
spp > pri; add pipe:0 ring:0 ring:1
spp > pri; add pipe:1 ring:2 ring:3
spp > pri; add pipe:2 ring:4 ring:5
spp > vf 1; component start fwd1 2 forward
spp > vf 1; component start fwd2 3 forward
```

(continues on next page)

(continued from previous page)

```
spp > vf 1; component start fwd3 4 forward
spp > vf 1; component start fwd4 5 forward
spp > vf 1; port add phy:0 rx fwd1
spp > vf 1; port add ring:0 tx fwd1
spp > vf 1; port add ring:1 rx fwd2
spp > vf 1; port add ring:2 tx fwd2
spp > vf 1; port add ring:3 rx fwd3
spp > vf 1; port add ring:4 tx fwd3
spp > vf 1; port add ring:5 rx fwd4
spp > vf 1; port add phy:1 tx fwd4
```

Since applications are connected not directly but through spp_vf, service chaining can be modified without restarting applications.

CHAPTER 5

SPP Commands

SPP provides commands for managing primary, secondary processes and SPP controller.

5.1 Primary Commands

Primary process is managed with `pri` command.

`pri` command takes a sub command. They must be separated with delimiter `;`. Some of sub commands take additional arguments.

```
spp > pri; SUB_CMD
```

All of Sub commands are referred with `help` command.

```
spp > help pri
Send a command to primary process.

    Show resources and statistics, or clear it.

        spp > pri; status  # show status

        spp > pri; clear   # clear statistics

    Launch secondary process..

        # Launch nfv:1
        spp > pri; launch nfv 1 -l 1,2 -m 512 -- -n 1 -s 192.168....

        # Launch vf:2
        spp > pri; launch vf 2 -l 1,4-7 -m 512 -- --client-id 2 -s ...
```

5.1.1 status

Show status fo `spp_primary` and forwarding statistics of each of ports.

```
spp > pri; status
- lcore_ids:
  - master: 0
- pipes:
  - pipe:0 ring:0 ring:1
- stats
  - physical ports:
      ID          rx          tx    tx_drop   rxq  txq mac_addr
       0           0           0           0   16   16 3c:fd:fe:b6:c4:28
       1           0           0           0 1024 1024 3c:fd:fe:b6:c4:29
       2           0           0           0    1    1 3c:fd:fe:b6:c4:30
  - ring ports:
      ID          rx          tx     rx_drop     tx_drop
       0           0           0           0           0
       1           0           0           0           0
       2           0           0           0           0
       ...
```

If you run `spp_primary` with forwarder thread, status of the forwarder is also displayed.

```
spp > pri; status
- lcore_ids:
  - master: 0
  - slave: 1
- forwarder:
  - status: idling
  - ports:
    - phy:0
    - phy:1
- pipes:
- stats
  - physical ports:
      ID          rx          tx    tx_drop  mac_addr
       0           0           0           0  56:48:4f:53:54:00
       1           0           0           0  56:48:4f:53:54:01
  - ring ports:
      ID          rx          tx     rx_drop     tx_drop
       0           0           0           0           0
       1           0           0           0           0
       ...
```

5.1.2 clear

Clear statistics.

```
spp > pri; clear
Clear port statistics.
```

5.1.3 add

Add a port with resource ID.

If the type of a port is other than pipe, specify port only. For example, adding `ring:0` by

```
spp > pri; add ring:0
Add ring:0.
```

Or adding `vhost:0` by

```
spp > pri; add vhost:0
Add vhost:0.
```

If the type of a port is pipe, specify a ring for rx and a ring for tx following a port. For example,

```
spp > pri; add pipe:0 ring:0 ring:1
Add pipe:0.
```

Note: pipe is independent of the forwarder and can be added even if the forwarder does not exist.

5.1.4 patch

Create a path between two ports, source and destination ports. This command just creates a path and does not start forwarding.

```
spp > pri; patch phy:0 ring:0
Patch ports (phy:0 -> ring:0).
```

5.1.5 forward

Start forwarding.

```
spp > pri; forward
Start forwarding.
```

Running status is changed from `idling` to `running` by executing it.

```
spp > pri; status
- lcore_ids:
  - master: 0
  - slave: 1
- forwarder:
  - status: running
  - ports:
    - phy:0
    - phy:1
...
```

5.1.6 stop

Stop forwarding.

```
spp > pri; stop
Stop forwarding.
```

Running status is changed from `running` to `idling` by executing it.

```
spp > pri; status
- lcore_ids:
  - master: 0
```

(continues on next page)

(continued from previous page)

```
  - slave: 1
- forwarder:
  - status: idling
  - ports:
    - phy:0
    - phy:1
...
```

5.1.7 del

Delete a port of given resource UID.

```
spp > pri; del ring:0
Delete ring:0.
```

5.1.8 launch

Launch a secondary process.

Spp_primary is able to launch a secondary process with given type, secondary ID and options of EAL and application itself. This is a list of supported type of secondary processes.

- nfv
- vf
- mirror
- pcap

```
# spp_nfv with sec ID 1
spp > pri; launch nfv 1 -l 1,2 -m 512 -- -n -s 192.168.1.100:6666

# spp_vf with sec ID 2
spp > pri; launch vf 2 -l 1,3-5 -m 512 -- --client-id -s 192.168.1.100:6666
```

You notice that `--proc-type secondary` is not given for launching secondary processes. `launch` command adds this option before requesting to launch the process so that you do not need to input this option by yourself.

`launch` command supports TAB completion for type, secondary ID and the rest of options. Some of EAL and application options are just a template, so you should edit them before launching. Some of default params of options, for instance, the number of lcores or the amount of memory, are changed from `config` command of *Common Commands*.

In terms of log, each of secondary processes are output its log messages to files under `log` directory of project root. The name of log file is defined with type of process and secondary ID. For instance, `nfv 2`, the path of log file is `log/spp_nfv-2.log`.

5.1.9 flow

Manipulate flow rules.

You can request `validate` before creating flow rule.

```
spp > pri; flow validate phy:0 ingress group 1 pattern eth dst is
      10:22:33:44:55:66 / vlan vid is 100 / end actions queue index 0 /
      of_pop_vlan / end
Flow rule validated
```

You can create rules by using `create` request.

```
spp > pri; flow create phy:0 ingress group 1 pattern eth dst is
      10:22:33:44:55:66 / vlan vid is 100 / end actions queue index 0 /
      of_pop_vlan / end
Flow rule #0 created
```

Note: `validate` and/or `create` in flow command tends to take long parameters. But you should enter it as one line. CLI assumes that new line means `command is entered`. So command should be entered without using new line.

You can delete specific flow rule.

```
spp > pri; flow destroy phy:0 0
Flow rule #0 destroyed
```

Listing flow rules per physical port is supported.

```
spp > pri; flow list phy:0
ID      Group   Prio    Attr    Rule
0       1       0       -e-     ETH => OF_PUSH_VLAN OF_SET_VLAN_VID OF_SET_VLAN_PCP
1       1       0       i--     ETH VLAN => QUEUE OF_POP_VLAN
2       0       0       i--     ETH => JUMP
```

The following is the parameters to be displayed.

- `ID`: Identifier of the rule which is unique per physical port.
- `Group`: Group number the rule belongs.
- `Prio`: Priority value of the rule.
- `Attr`: Attributes for the rule which is independent each other. The possible values of `Attr` are `i` or `e` or `t`. `i` means ingress. `e` means egress and `t` means transfer.
- `Rule`: Rule notation.

Flow detail can be listed.

```
spp > pri; flow status phy:0 0
Attribute:
  Group   Priority Ingress Egress Transfer
  1       0        true    false  false
Patterns:
  - eth:
    - spec:
      - dst: 10:22:33:44:55:66
      - src: 00:00:00:00:00:00
      - type: 0xffff
    - last:
    - mask:
      - dst: ff:ff:ff:ff:ff:ff
      - src: 00:00:00:00:00:00
```

(continues on next page)

(continued from previous page)

```
      - type: 0xffff
  - vlan:
    - spec:
      - tci: 0x0064
      - inner_type: 0x0000
    - last:
    - mask:
      - tci: 0xffff
      - inner_type: 0x0000
Actions:
  - queue:
    - index: 0
  - of_pop_vlan:
```

5.2 Secondary Commands

5.2.1 spp_nfv

Each of `spp_nfv` and `spp_vm` processes is managed with `nfv` command. It is for sending sub commands to secondary with specific ID called secondary ID.

`nfv` command takes an secondary ID and a sub command. They must be separated with delimiter `;`. Some of sub commands take additional arguments for speicfying resource owned by secondary process.

```
spp > nfv SEC_ID; SUB_CMD
```

All of Sub commands are referred with `help` command.

```
spp > help nfv

Send a command to secondary process specified with ID.

    SPP secondary process is specified with secondary ID and takes
    sub commands.

    spp > nfv 1; status
    spp > nfv 1; add ring:0
    spp > nfv 1; patch phy:0 ring:0

    You can refer all of sub commands by pressing TAB after
    'nfv 1;'.

    spp > nfv 1;  # press TAB
    add     del     exit    forward patch   status  stop
```

status

Show running status and ports assigned to the process. If a port is patched to other port, source and destination ports are shown, or only source if it is not patched.

```
spp > nfv 1; status
- status: idling
```

(continues on next page)

(continued from previous page)

```
- lcores: [1, 2]
- ports:
  - phy:0 -> ring:0
  - phy:1
```

add

Add a port to the secondary with resource ID.

For example, adding `ring:0` by

```
spp > nfv 1; add ring:0
Add ring:0.
```

Or adding `vhost:0` by

```
spp > nfv 1; add vhost:0
Add vhost:0.
```

patch

Create a path between two ports, source and destination ports. This command just creates a path and does not start forwarding.

```
spp > nfv 1; patch phy:0 ring:0
Patch ports (phy:0 -> ring:0).
```

forward

Start forwarding.

```
spp > nfv 1; forward
Start forwarding.
```

Running status is changed from `idling` to `running` by executing it.

```
spp > nfv 1; status
- status: running
- ports:
  - phy:0
  - phy:1
```

stop

Stop forwarding.

```
spp > nfv 1; stop
Stop forwarding.
```

Running status is changed from `running` to `idling` by executing it.

```
spp > nfv 1; status
- status: idling
- ports:
  - phy:0
  - phy:1
```

del

Delete a port from the secondary.

```
spp > nfv 1; del ring:0
Delete ring:0.
```

exit

Terminate the secondary. For terminating all secondaries, use `bye sec` command instead of it.

```
spp > nfv 1; exit
```

5.2.2 spp_vf

`spp_vf` is a kind of SPP secondary process. It is introduced for providing SR-IOV like features.

Each of `spp_vf` processes is managed with `vf` command. It is for sending sub commands with specific ID called secondary ID for changing configuration, assigning or releasing resources.

Secondary ID is referred as `--client-id` which is given as an argument while launching `spp_vf`. It should be unique among all of secondary processes including `spp_nfv` and others.

`vf` command takes an secondary ID and one of sub commands. Secondary ID and sub command should be separated with delimiter `;`, or failed to a command error. Some of sub commands take additional arguments for configuration of the process or its resource management.

```
spp > vf SEC_ID; SUB_CMD
```

In this example, `SEC_ID` is a secondary ID and `SUB_CMD` is one of the following sub commands. Details of each of sub commands are described in the next sections.

- status
- component
- port
- classifier_table

`spp_vf` supports TAB completion. You can complete all of the name of commands and its arguments. For instance, you find all of sub commands by pressing TAB after `vf SEC_ID;`.

```
spp > vf 1;  # press TAB key
classifier_table  component      port        status
```

It tries to complete all of possible arguments. However, `spp_vf` takes also an arbitrary parameter which cannot be predicted, for example, name of component MAC address. In this cases, `spp_vf` shows capitalized keyword for indicating it is an arbitrary parameter. Here is an exmaple of `component` command to initialize a worker thread. Keyword `NAME` should be replaced with your favorite name for the worker of the role.

```
spp > vf 1; component st  # press TAB key to show args starting 'st'
start  stop
spp > vf 1; component start NAME  # 'NAME' is shown with TAB after start
spp > vf 1; component start fw1   # replace 'NAME' with your favorite name
spp > vf 1; component start fw1   # then, press TAB to show core IDs
5  6  7  8
```

It is another example of replacing keyword. `port` is a sub command for assigning a resource to a worker thread. `RES_UID` is replaced with resource UID which is a combination of port type and its ID such as `ring:0` or `vhost:1` to assign it as RX port of forwarder `fw1`.

```
spp > vf 1; port add RES_UID
spp > vf 1; port add ring:0 rx fw1
```

If you are reached to the end of arguments, no candidate keyword is displayed. It is a completed statement of `component` command, and TAB completion does not work after `forward` because it is ready to run.

```
spp > vf 1; component start fw1 5 forward
Succeeded to start component 'fw1' on core:5
```

It is also completed secondary IDs of `spp_vf` and it is helpful if you run several `spp_vf` processes.

```
spp > vf  # press TAB after space following 'vf'
1;  3;    # you find two spp_vf processes of sec ID 1, 3
```

By the way, it is also a case of no candidate keyword is displayed if your command statement is wrong. You might be encountered an error if you run the wrong command. Please take care.

```
spp > vf 1; compo  # no candidate shown for wrong command
Invalid command "compo".
```

status

Show the information of worker threads and its resources. Status information consists of three parts.

```
spp > vf 1; status
Basic Information:
 - client-id: 1
 - ports: [phy:0 nq 0, phy:0 nq 1, ring:0, ring:1, ring:2]
 - lcore_ids:
   - master: 1
   - slaves: [2, 3 ,4, 5]
Classifier Table:
 - C0:8E:CD:38:EA:A8, ring:1
 - C0:8E:CD:38:BC:E6, ring:2
Components:
 - core:2 'fwd1' (type: forward)
```

(continues on next page)

(continued from previous page)

```
  - rx: phy:0 nq 0
  - tx: ring:0
- core:3 'mg' (type: merge)
- core:4 'cls' (type: classifier)
  - rx: phy:0 nq 1
  - tx: ring:1
  - tx: ring:2
- core:5 '' (type: unuse)
```

`Basic Information` is for describing attributes of `spp_vf` itself. `client-id` is a secondary ID of the process and `ports` is a list of all of ports owned the process.

`Classifier Table` is a list of entries of `classifier` worker thread. Each of entry is a combination of MAC address and destination port which is assigned to this thread.

`Components` is a list of all of worker threads. Each of workers has a core ID running on, type of the worker and a list of resources. Entry of no name with `unuse` type means that no worker thread assigned to the core. In other words, it is ready to be assigned.

component

Assign or release a role of forwarding to worker threads running on each of cores which are reserved with `-c` or `-l` option while launching `spp_vf`. The role of the worker is chosen from `forward`, `merge` or `classifier`.

`forward` role is for simply forwarding from source port to destination port. On the other hands, `merge` role is for receiving packets from multiple ports as N:1 communication, or `classifier` role is for sending packet to multiple ports by referring MAC address as 1:N communication.

You are required to give an arbitrary name with as an ID for specifying the role. This name is also used while releasing the role.

```
# assign 'ROLE' to worker on 'CORE_ID' with a 'NAME'
spp > vf SEC_ID; component start NAME CORE_ID ROLE

# release worker 'NAME' from the role
spp > vf SEC_ID; component stop NAME
```

Here are some examples of assigning roles with `component` command.

```
# assign 'forward' role with name 'fw1' on core 2
spp > vf 2; component start fw1 2 forward

# assign 'merge' role with name 'mgr1' on core 3
spp > vf 2; component start mgr1 3 merge

# assign 'classifier' role with name 'cls1' on core 4
spp > vf 2; component start cls1 4 classifier
```

In the above examples, each different `CORE-ID` is specified to each role. You can assign several components on the same core, but performance might be decreased. This is an example for assigning two roles of `forward` and `merge` on the same `core 2`.

```
# assign two roles on the same 'core 2'.
spp > vf 2; component start fw1 2 forward
spp > vf 2; component start mgr1 2 merge
```

Examples of releasing roles.

```
# release roles
spp > vf 2; component stop fw1
spp > vf 2; component stop mgr1
spp > vf 2; component stop cls1
```

port

Add or delete a port to a worker.

Adding port

```
spp > vf SEC_ID; port add RES_UID [nq QUEUE_NUM] DIR NAME
```

`RES_UID` is with replaced with resource UID such as `ring:0` or `vhost:1`. `spp_vf` supports three types of port. `nq QUEUE_NUM` is the queue number when multi-queue is configured. This is optional parameter.

- `phy` : Physical NIC
- `ring` : Ring PMD
- `vhost` : Vhost PMD

`DIR` means the direction of forwarding and it should be `rx` or `tx`. `NAME` is the same as for `component` command.

This is an example for adding ports to a classifer `cls1`. In this case, it is configured to receive packets from `phy:0` and send it to `ring:0` or `ring:1`. The destination is decided with MAC address of the packets by referring the table. How to configure the table is described in *classifier_table* command.

```
# recieve from 'phy:0'
spp > vf 2; port add phy:0 rx cls1

# receive from queue 1 of 'phy:0'
spp > vf 2; port add phy:0 nq 1 rx cls1

# send to 'ring:0' and 'ring:1'
spp > vf 2; port add ring:0 tx cls1
spp > vf 2; port add ring:1 tx cls1
```

`spp_vf` also supports VLAN features, adding or deleting VLAN tag. It is used remove VLAN tags from incoming packets from outside of host machine, or add VLAN tag to outgoing packets.

To configure VLAN features, use additional sub command `add_vlantag` or `del_vlantag` followed by `port` sub command.

To remove VLAN tag, simply add `del_vlantag` sub command without arguments.

```
spp > vf SEC_ID; port add RES_UID [nq QUEUE_NUM] DIR NAME del_vlantag
```

On the other hand, use `add_vlantag` which takes two arguments, `VID` and `PCP`, for adding VLAN tag to the packets.

```
spp > vf SEC_ID; port add RES_UID [nq QUEUE_NUM] DIR NAME add_vlantag VID PCP
```

`VID` is a VLAN ID and `PCP` is a Priority Code Point defined in IEEE 802.1p. It is used for QoS by defining priority ranged from lowest prioroty `0` to the highest `7`.

Here is an example of use of VLAN features considering a use case of a forwarder removes VLAN tag from incoming packets and another forwarder adds VLAN tag before sending packet outside.

```
# remove VLAN tag in forwarder 'fw1'
spp > vf 2; port add phy:0 rx fw1 del_vlantag

# add VLAN tag with VLAN ID and PCP in forwarder 'fw2'
spp > vf 2; port add phy:1 tx fw2 add_vlantag 101 3
```

Adding port may cause component to start packet forwarding. Please see detail in *design spp_vf*.

Until one rx port and one tx port are added, forwarder does not start packet forwarding. If it is requested to add more than one rx and one tx port, it replies an error message. Until at least one rx port and two tx ports are added, classifier does not start packet forwarding. If it is requested to add more than two rx ports, it replies an error message. Until at least two rx ports and one tx port are added, merger does not start packet forwarding. If it is requested to add more than two tx ports, it replies an error message.

Deleting port

Delete a port which is not used anymore.

```
spp > vf SEC_ID; port del RES_UID [nq QUEUE_NUM] DIR NAME
```

It is same as the adding port, but no need to add additional sub command for VLAN features.

Here is an example.

```
# delete rx port 'ring:0' from 'cls1'
spp > vf 2; port del ring:0 rx cls1

# delete rx port queue 1 of 'phy:0' from 'cls1'
spp > vf 2; port del phy:0 nq 1 rx cls1

# delete tx port 'vhost:1' from 'mgr1'
spp > vf 2; port del vhost:1 tx mgr1
```

Note: Deleting port may cause component to stop packet forwarding. Please see detail in *design spp_vf*.

classifier_table

Register an entry of a combination of MAC address and port to a table of classifier.

```
# add entry
spp > vf SEC_ID; classifier_table add mac MAC_ADDR RES_UID

# delete entry
spp > vf SEC_ID; classifier_table del mac MAC_ADDRESS RES_ID

# add entry with multi-queue support
spp > vf SEC_ID; classifier_table add mac MAC_ADDR RES_UID [nq QUEUE_NUM]

# delete entry with multi-queue support
spp > vf SEC_ID; classifier_table del mac MAC_ADDRESS RES_ID [nq QUEUE_NUM]
```

This is an example to register MAC address `52:54:00:01:00:01` with port `ring:0`.

```
spp > vf 1; classifier_table add mac 52:54:00:01:00:01 ring:0
```

Classifier supports the `default` entry for packets which does not match any of entries in the table. If you assign `ring:1` as default, simply specify `default` instead of MAC address.

```
spp > vf 1; classifier_table add mac default ring:1
```

`classifier_table` sub command also supports VLAN features as similar to `port`.

```
# add entry with VLAN features
spp > vf SEC_ID; classifier_table add vlan VID MAC_ADDR RES_UID

# delete entry of VLAN
spp > vf SEC_ID; classifier_table del vlan VID MAC_ADDR RES_UID
```

Here is an example for adding entries.

```
# add entry with VLAN tag
spp > vf 1; classifier_table add vlan 101 52:54:00:01:00:01 ring:0

# add entry of default with VLAN tag
spp > vf 1; classifier_table add vlan 101 default ring:1
```

Delete an entry. This is an example to delete an entry with VLAN tag 101.

```
# delete entry with VLAN tag
spp > vf 1; classifier_table del vlan 101 52:54:00:01:00:01 ring:0
```

exit

Terminate the spp_vf.

```
spp > vf 1; exit
```

5.2.3 spp_mirror

`spp_mirror` is an implementation of TaaS feature as a SPP secondary process for port mirroring.

Each of `spp_mirror` processes is managed with `mirror` command. Because basic design of `spp_mirror` is derived from `spp_vf`, its commands are almost similar to `spp_vf`.

Secondary ID is referred as `--client-id` which is given as an argument while launching `spp_mirror`. It should be unique among all of secondary processes including `spp_nfv` and others.

`mirror` command takes an secondary ID and one of sub commands. Secondary ID and sub command should be separated with delimiter `;`, or failed to a command error. Some of sub commands take additional arguments for configuration of the process or its resource management.

```
spp > mirror SEC_ID; SUB_CMD
```

In this example, `SEC_ID` is a secondary ID and `SUB_CMD` is one of the following sub commands. Details of each of sub commands are described in the next sections.

- status
- component
- port

`spp_mirror` supports TAB completion. You can complete all of the name of commands and its arguments. For instance, you find all of sub commands by pressing TAB after `mirror SEC_ID;`.

```
spp > mirror 1;  # press TAB key
component      port        status
```

It tries to complete all of possible arguments. However, `spp_mirror` takes also an arbitrary parameter which cannot be predicted, for example, name of component. In this cases, `spp_mirror` shows capitalized keyword for indicating it is an arbitrary parameter. Here is an exmaple of `component` command to initialize a worker thread. Keyword `NAME` should be replaced with your favorite name for the worker of the role.

```
spp > mirror 1; component st  # press TAB key to show args starting 'st'
start  stop
spp > mirror 1; component start NAME  # 'NAME' is shown with TAB after start
spp > mirror 1; component start mr1   # replace 'NAME' with favorite name
spp > mirror 1; component start mr1   # then, press TAB to show core IDs
5  6  7
```

It is another example of replacing keyword. `port` is a sub command for assigning a resource to a worker thread. `RES_UID` is replaced with resource UID which is a combination of port type and its ID such as `ring:0` or `vhost:1` to assign it as RX port of forwarder `mr1`.

```
spp > mirror 1; port add RES_UID
spp > mirror 1; port add ring:0 rx mr1
```

If you are reached to the end of arguments, no candidate keyword is displayed. It is a completed statement of `component` command, and TAB completion does not work after `mirror` because it is ready to run.

```
spp > mirror 1; component start mr1 5 mirror
Succeeded to start component 'mr1' on core:5
```

It is also completed secondary IDs of `spp_mirror` and it is helpful if you run several `spp_mirror` processes.

```
spp > mirror  # press TAB after space following 'mirror'
1;  3;    # you find two spp_mirror processes of sec ID 1, 3
```

By the way, it is also a case of no candidate keyword is displayed if your command statement is wrong. You might be encountered an error if you run the wrong command. Please take care.

```
spp > mirror 1; compa  # no candidate shown for wrong command
Invalid command "compa".
```

status

Show the information of worker threads and its resources. Status information consists of three parts.

```
spp > mirror 1; status
Basic Information:
  - client-id: 3
  - ports: [phy:0, phy:1, ring:0, ring:1, ring:2, ring:3, ring:4]
 - lcore_ids:
   - master: 1
   - slaves: [2, 3, 4]
Components:
  - core:5 'mr1' (type: mirror)
    - rx: ring:0
    - tx: [ring:1, ring:2]
  - core:6 'mr2' (type: mirror)
    - rx: ring:3
    - tx: [ring:4, ring:5]
  - core:7 '' (type: unuse)
```

`Basic Information` is for describing attributes of `spp_mirror` itself. `client-id` is a secondary ID of the process and `ports` is a list of all of ports owned the process.

`Components` is a list of all of worker threads. Each of workers has a core ID running on, type of the worker and a list of resources. Entry of no name with `unuse` type means that no worker thread assigned to the core. In other words, it is ready to be assinged.

component

Assing or release a role of forwarding to worker threads running on each of cores which are reserved with `-c` or `-l` option while launching `spp_mirror`. Unlike `spp_vf`, `spp_mirror` only has one role, `mirror`.

```
# assign 'ROLE' to worker on 'CORE_ID' with a 'NAME'
spp > mirror SEC_ID; component start NAME CORE_ID ROLE

# release worker 'NAME' from the role
spp > mirror SEC_ID; component stop NAME
```

Here is an example of assigning role with `component` command.

```
# assign 'mirror' role with name 'mr1' on core 2
spp > mirror 2; component start mr1 2 mirror
```

And an examples of releasing role.

```
# release mirror role
spp > mirror 2; component stop mr1
```

port

Add or delete a port to a worker.

Adding port

```
spp > mirror SEC_ID; port add RES_UID DIR NAME
```

`RES_UID` is with replaced with resource UID such as `ring:0` or `vhost:1`. `spp_mirror` supports three types of port.

- `phy` : Physical NIC
- `ring` : Ring PMD
- `vhost` : Vhost PMD

`DIR` means the direction of forwarding and it should be `rx` or `tx`. `NAME` is the same as for `component` command.

This is an example for adding ports to `mr1`. In this case, it is configured to receive packets from `ring:0` and send it to `vhost:0` and `vhost:1` by duplicating the packets.

```
# recieve from 'phy:0'
spp > mirror 2; port add ring:0 rx mr1

# send to 'ring:0' and 'ring:1'
spp > mirror 2; port add vhost:0 tx mr1
spp > mirror 2; port add vhost:1 tx mr1
```

Adding port may cause component to start packet forwarding. Please see details in *design spp_mirror*.

Until one rx and two tx ports are registered, `spp_mirror` does not start forwarding. If it is requested to add more than one rx and two tx ports, it replies an error message.

Deleting port

Delete a port which is not be used anymore. It is almost same as adding port.

```
spp > mirror SEC_ID; port del RES_UID DIR NAME
```

Here is some examples.

```
# delete rx port 'ring:0' from 'mr1'
spp > mirror 2; port del ring:0 rx mr1

# delete tx port 'vhost:1' from 'mr1'
spp > mirror 2; port del vhost:1 tx mr1
```

Note: Deleting port may cause component to stop packet forwarding. Please see detail in *design spp_mirror*.

exit

Terminate `spp_mirror` process.

```
spp > mirror 2; exit
```

5.2.4 spp_pcap

`spp_pcap` is a kind of SPP secondary process. It it introduced for providing packet capture features.

Each of `spp_pcap` processes is managed with `pcap` command. It is for sending sub commands with specific ID called secondary ID for starting or stopping packet capture.

Secondary ID is referred as `--client-id` which is given as an argument while launching `spp_pcap`. It should be unique among all of secondary processes including `spp_nfv`, `spp_vm` and others.

`pcap` command takes an secondary ID and one of sub commands. Secondary ID and sub command should be separated with delimiter `;`, or failed to a command error.

```
spp > pcap SEC_ID; SUB_CMD
```

In this example, `SEC_ID` is a secondary ID and `SUB_CMD` is one of the following sub commands. Details of each of sub commands are described in the next sections.

- status
- start
- stop
- exit

`spp_pcap` supports TAB completion. You can complete all of the name of commands and its arguments. For instance, you find all of sub commands by pressing TAB after `pcap SEC_ID;`.

```
spp > pcap 1;  # press TAB key
exit  start      status        stop
```

It tries to complete all of possible arguments.

```
spp > pcap 1; component st  # press TAB key to show args starting 'st'
start  status  stop
```

If you are reached to the end of arguments, no candidate keyword is displayed. It is a completed statement of `start` command, and TAB completion does not work after `start` because it is ready to run.

```
spp > pcap 1; start
Succeeded to start capture
```

It is also completed secondary IDs of `spp_pcap` and it is helpful if you run several `spp_pcap` processes.

```
spp > pcap  # press TAB after space following 'pcap'
1;  3;    # you find two spp_pcap processes of sec ID 1, 3
```

By the way, it is also a case of no candidate keyword is displayed if your command statement is wrong. You might be encountered an error if you run the wrong command. Please take care.

```
spp > pcap 1; ste  # no candidate shown for wrong command
Invalid command "ste".
```

status

Show the information of worker threads of `receiver` and `writer` threads and its resources.

```
spp > pcap 1; status
Basic Information:
  - client-id: 1
  - status: idling
  - lcore_ids:
    - master: 1
    - slaves: [2, 3, 4, 5, 6]
Components:
  - core:2 receive
    - rx: phy:0
  - core:3 write
    - filename:
  - core:4 write
    - filename:
  - core:5 write
    - filename:
  - core:6 write
    - filename:
```

`client-id` is a secondary ID of the process and `status` shows running status.

Each of lcore has a role of `receive` or `write`. `receiver` has capture port as input and `write` has a capture file as output, but the `filename` is empty while `idling` status because capturing is not started yet.

If you start capturing, you can find each of `writer` threads has a capture file. After capturing is stopped, `filename` is returned to be empty again.

```
spp > pcap 2; status
  - client-id: 2
  - status: running
  - core:2 receive
    - rx: phy:0
  - core:3 write
    - filename: /tmp/spp_pcap.20190214161550.phy0.1.1.pcap.lz4
  - core:4 write
    - filename: /tmp/spp_pcap.20190214161550.phy0.2.1.pcap.lz4
  - core:5 write
    - filename: /tmp/spp_pcap.20190214161550.phy0.3.1.pcap.lz4
  - core:6 write
    - filename: /tmp/spp_pcap.20190214161550.phy0.4.1.pcap.lz4
```

start

Start packet capture.

```
# start capture
spp > pcap SEC_ID; start
```

Here is a example of starting capture.

```
# start capture
spp > pcap 1; start
Start packet capture.
```

stop

Stop packet capture.

```
# stop capture
spp > pcap SEC_ID; stop
```

Here is a example of stopping capture.

```
# stop capture
spp > pcap 2; stop
Start packet capture.
```

exit

Terminate the `spp_pcap`.

```
spp > pcap 1; exit
```

5.3 Common Commands

5.3.1 status

Show the status of SPP processes.

```
spp > status
- spp-ctl:
  - address: 172.30.202.151:7777
- primary:
  - status: running
- secondary:
  - processes:
    1: nfv:1
    2: vf:3
```

5.3.2 config

Show or update config params.

Config params used for changing behaviour of SPP CLI. For instance, if you change command prompt, you can set any of prompt with `config` command other than default `spp >`.

```
# set prompt to "$ spp "
spp > config prompt "$ spp "
Set prompt: "$ spp "
$ spp
```

Show Config

To show the list of config all of params, simply run `config`.

```
# show list of config
spp > config
- max_secondary: "16"       # The maximum number of secondary processes
- sec_nfv_nof_lcores: "1"   # Default num of lcores for workers of spp_nfv
- topo_size: "60%"  # Percentage or ratio of topo
- sec_base_lcore: "1"       # Shared lcore among secondaries
....
```

Or show params only started from `sec_`, add the keyword to the commnad.

```
# show config started from `sec_`
spp > config sec_
- sec_vhost_cli: "" # Vhost client mode, activated if set any of values
- sec_mem: "-m 512" # Mem size
- sec_nfv_nof_lcores: "1"   # Default num of lcores for workers of spp_nfv
- sec_base_lcore: "1"       # Shared lcore among secondaryes
....
```

Set Config

One of typical uses of `config` command is to change the default params of other commands. `pri; launch` takes several options for launching secondary process and it is completed by using default params started from `sec_`.

```
spp > pri; launch nfv 2  # press TAB for completion
spp > pri; launch nfv 2 -l 1,2 -m 512 -- -n 2 -s 192.168.11.59:6666
```

The default number of memory size is `-m 512` and the definition `sec_mem` can be changed with `config` command. Here is an example of changing `-m 512` to `--socket-mem 512,0`.

```
spp > config sec_mem "--socket-mem 512,0"
Set sec_mem: "--socket-mem 512,0"
```

After updating the param, expanded options is also updated.

```
spp > pri; launch nfv 2  # press TAB for completion
spp > pri; launch nfv 2 -l 1,2 --socket-mem 512,0 -- -n 2 -s ...
```

5.3.3 playback

Restore network configuration from a recipe file which defines a set of SPP commands. You can prepare a recipe file by using `record` command or editing file by hand.

It is recommended to use extension `.rcp` to be self-sxplanatory as a recipe, although you can use any of extensions such as `.txt` or `.log`.

```
spp > playback /path/to/my.rcp
```

5.3.4 record

Start recording user's input and create a recipe file for loading from `playback` commnad. Recording recipe is stopped by executing `exit` or `playback` command.

```
spp > record /path/to/my.rcp
```

Note: It is not supported to stop recording without `exit` or `playback` command. It is planned to support `stop` command for stopping record in next relase.

5.3.5 history

Show command history. Command history is recorded in a file named `$HOME/.spp_history`. It does not add some command which are no meaning for history, `bye`, `exit`, `history` and `redo`.

```
spp > history
  1  ls
  2  cat file.txt
```

5.3.6 redo

Execute command of index of history.

```
spp > redo 5  # exec 5th command in the history
```

5.3.7 server

Switch SPP REST API server.

SPP CLI is able to manage several SPP nodes via REST API servers. It is also abaivable to register new one, or unregister.

Show all of registered servers by running `server list` or simply `server`. Notice that `*` means that the first node is under the control of SPP CLI.

```
spp > server
  1: 192.168.1.101:7777 *
  2: 192.168.1.102:7777

spp > server list  # same as above
  1: 192.168.1.101:7777 *
  2: 192.168.1.102:7777
```

Switch to other server by running `server` with index or address displayed in the list. Port number can be omitted if it is default `7777`.

```
# Switch to the second node
spp > server 2
Switch spp-ctl to "2: 192.168.1.102:7777".

# Switch to firt one again with address
spp > server 192.168.1.101  # no need for port num
Switch spp-ctl to "1: 192.168.1.101:7777".
```

Register new one by using `add` command, or unregister by `del` command with address. For unregistering, node is also specified with index.

```
# Register
spp > server add 192.168.122.177
Registered spp-ctl "192.168.122.177:7777".

# Unregister second one
spp > server del 2  # or 192.168.1.102
Unregistered spp-ctl "192.168.1.102:7777".
```

You cannot unregister node under the control, or switch to other one before.

```
spp > server del 1
Cannot del server "1" in use!
```

5.3.8 env

Show environmental variables. It is mainly used to find variables related to SPP.

```
# show all env varibles.
spp > env

# show env varibles starts with `SPP`.
spp > env SPP
```

5.3.9 pwd

Show current path.

```
spp> pwd
/path/to/curdir
```

5.3.10 cd

Change current directory.

```
spp> cd /path/to/dir
```

5.3.11 ls

Show a list of directory contents.

```
spp> ls /path/to/dir
```

5.3.12 mkdir

Make a directory.

```
spp> mkdir /path/to/dir
```

5.3.13 cat

Show contents of a file.

```
spp> cat /path/to/file
```

5.3.14 less

Show contents of a file.

```
spp> less /path/to/file
```

5.3.15 bye

`bye` command is for terminating SPP processes. It supports two types of termination as sub commands.

- sec
- all

First one is for terminating only secondary processes at once.

```
spp > bye sec
Closing secondary ...
Exit nfv 1
Exit vf 3.
```

Second one is for all SPP processes other than controller.

```
spp > bye all
Closing secondary ...
Exit nfv 1
Exit vf 3.
Closing primary ...
Exit primary
```

5.3.16 exit

Same as `bye` command but just for terminating SPP controller and not for other processes.

```
spp > exit
Thank you for using Soft Patch Panel
```

5.3.17 help

Show help message for SPP commands.

```
spp > help

Documented commands (type help <topic>):
========================================
bye  exit     inspect   ls      nfv       pwd     server  topo_resize
cat  help     less      mirror  playback  record  status  topo_subgraph
cd   history  load_cmd  mkdir   pri       redo    topo    vf

spp > help status
Display status info of SPP processes

    spp > status

spp > help nfv
Send a command to spp_nfv specified with ID.

    Spp_nfv is specified with secondary ID and takes sub commands.

    spp > nfv 1; status
    spp > nfv 1; add ring:0
    spp > nfv 1; patch phy:0 ring:0

    You can refer all of sub commands by pressing TAB after
    'nfv 1;'.

    spp > nfv 1;  # press TAB
    add     del     exit    forward patch   status  stop
```

5.4 Experimental Commands

There are experimental commands in SPP controller. It might not work for some cases properly because it is not well tested currently.

5.4.1 topo

Output network topology in several formats. Support four types of output.

- Terminal
- Browser (websocket server is required)
- Text (dot, json, yaml)
- Image file (jpg, png, bmp)

This command uses graphviz for generating topology file. You can also generate a dot formatted file or image files supported by graphviz.

Here is an example for installing required tools for `topo term` command to output in a terminal.

```
$ sudo apt install graphviz \
  imagemagick \
  libsixel-bin
```

MacOS is also supported optionally for using topo runs on a remote host. In this case, iTerm2 and imgcat are required as described in the next section.

To output in browser with `topo http` command, install required packages by using `requirements.txt` as described in *install SPP*, or only for them as follwoing.

```
$ pip3 install tornado \
  websocket-client
```

Output to Terminal

Output an image of network configuration in terminal.

```
spp > topo term
```

There are few terminal applications supporting to output image with `topo`. You can use mlterm, xterm or other terminals supported by img2sixel. You can also use iTerm2 on MacOS. If you use iTerm2, you need to downloada a shell script `imgcat` from iTerm2's displaying support site and save this script as `src/controller/3rd_party/imgcat` with permission `775`. `topo` command tries to `img2sixel` first, then `imgcat` in the `3rd_party` directory.

Fig. 5.1: topo term example

Output to Browser

Output an image of network configuration in browser.

```
spp > topo http
```

`topo term` is useful to understand network configuration intuitively. However, it should be executed on a node running SPP controller. You cannnot see the image if you login remote node via ssh and running SPP controller on remote.

Websocket server is launched from `src/controller/websocket/spp_ws.py` to accept client messages. You should start it before using `topo term` command. Then, open url shown in the terminal (default is `http://127.0.0.1:8989`).

Browser and SPP controller behave as clients, but have different roles. Browser behaves as a viwer and SPP controller behaves as a udpater. If you update network configuration and run `topo http` command, SPP controller sends a message containing network configuration as DOT language format. Once the message is accepted, websocket server sends it to viewer clients immediately.

Output to File

Output a text or image of network configuration to a file.

```
spp > topo [FILE_NAME] [FILE_TYPE]
```

You do not need to specify `FILE_TYPE` because `topo` is able to decide file type from `FILE_NAME`. It is optional. This is a list of supported file type.

- dot
- js (or json)
- yml (or yaml)
- jpg
- png
- bmp

To generate a DOT file `network.dot`, run `topo` command with file name.

```
# generate DOT file
spp > topo network.dot
Create topology: 'network.dot'
# show contents of the file
spp > cat network.dot
digraph spp{
newrank=true;
node[shape="rectangle", style="filled"];
...
```

To generate a jpg image, run `topo` with the name `network.jpg`.

```
spp > topo network.jpg
spp > ls
...  network.jpg  ...
```

5.4.2 topo_subgraph

`topo_subgraph` is a supplemental command for managing subgraphs for `topo`.

```
spp > topo_subgraph VERB LABEL RES_ID1,RES_ID2,...
```

Each of options are:

- VERB: `add` or `del`
- LABEL: Arbitrary text, such as `guest_vm1` or `container1`
- RES_ID: Series of Resource ID consists of type and ID such as `vhost:1`. Each of resource IDs are separated with `,` or `;`.

Subgraph is a group of object defined in dot language. Grouping objects helps your understanding relationship or hierarchy of each of objects. It is used for grouping resources on VM or container to be more understandable.

For example, if you create two vhost interfaces for a guest VM and patch them to physical ports, `topo term` shows a network configuration as following.

Fig. 5.2: Before using topo_subgraph

Two of vhost interfaces are placed outside of `Host` while the guest VM runs on `Host`. However, `vhost:1` and `vhost:2` should be placed inside `Host` actually. It is required to use subgraph!

To include guest VM and its resources inside the `Host`, use `topo_subgraph` with options. In this case, add subgraph `guest_vm` and includes resoures `vhost:1` and `vhost:2` into the subgraph.

```
spp > topo_subgraph add guest_vm vhost:1,vhost:2
```

Fig. 5.3: After using topo_subgraph

All of registered subgraphs are listed by using `topo_subgraph` with no options.

```
spp > topo_subgraph
label: guest_vm subgraph: "vhost:1,vhost:2"
```

If guest VM is shut down and subgraph is not needed anymore, delete subgraph `guest_vm`.

```
spp > topo_subgraph del guest_vm
```

5.4.3 load_cmd

Load command plugin dynamically while running SPP controller.

```
spp > load_cmd [CMD_NAME]
```

CLI of SPP controller is implemented with `Shell` class which is derived from Python standard library `Cmd`. It means that subcommands of SPP controller must be implemented as a member method named as `do_xxx`. For instance, `status` subcommand is implemented as `do_status` method.

`load_cmd` is for providing a way to define user specific command as a plugin. Plugin file must be placed in `spp/src/controller/command` and command name must be the same as file name. In addition, `do_xxx` method must be defined which is called from SPP controller.

For example, `hello` sample plugin is defined as `spp/src/controller/command/hello.py` and `do_hello` is defined in this plugin. Comment for `do_hello` is used as help message for `hello` command.

```
def do_hello(self, name):
    """Say hello to given user

    spp > hello alice
    Hello, alice!
    """

    if name == '':
        print('name is required!')
    else:
        hl = Hello(name)
        hl.say()
```

`hello` is loaded and called as following.

```
spp > load_cmd hello
Module 'command.hello' loaded.
spp > hello alice
Hello, alice!
```

CHAPTER 6

Tools

6.1 SPP Container

Running SPP and DPDK applications on containers.

6.1.1 Overview

SPP container is a set of tools for running SPP and DPDK applications with docker. It consists of shell or python scripts for building container images and launching app containers with docker commands.

As shown in Fig. 6.1, all of DPDK applications, including SPP primary and secondary processes, run inside containers. SPP controller is just a python script and does not need to be run as an app container.

Fig. 6.1: SPP container overview

6.1.2 Getting Started

In this section, learn how to use SPP container with a simple usecase. You use four of terminals for running SPP processes and applications.

Setup DPDK and SPP

First of all, you need to clone DPDK and setup hugepages for running DPDK application as described in *Setup* or DPDK's Gettting Started Guide. You also need to load kernel modules and bind network ports as in Linux Drivers.

Then, as described in *Install DPDK and SPP* , clone and compile SPP in any directory.

```
# Terminal 1
$ git clone http://dpdk.org/git/apps/spp
$ cd spp
```

Build Docker Images

Build tool is a python script for creating a docker image and currently supporting three types of images for DPDK sample applications, pktgen-dpdk, or SPP.

Run build tool for creating three type of docker images. It starts to download the latest Ubuntu docker image and installation for the latest DPDK, pktgen or SPP.

```
# Terminal 1
$ cd /path/to/spp/tools/sppc
$ python3 build/main.py -t dpdk
$ python3 build/main.py -t pktgen
$ python3 build/main.py -t spp
```

Of course DPDK is required from pktgen and SPP, and it causes a problem of compatibility between them sometimes. In this case, you should build SPP with `--dpdk-branch` option to tell the version of DPDK explicitly.

```
# Terminal 1
$ python3 build/main.py -t spp --dpdk-branch v19.11
```

You can find all of options by `build/main.py -h`.

Waiting for a minutes, then you are ready to launch app containers. All of images are referred from `docker images` command.

```
$ docker images
REPOSITORY            TAG       IMAGE ID         CREATED          SIZE
sppc/spp-ubuntu       latest    3ec39adb460f     2 days ago       862MB
sppc/pktgen-ubuntu    latest    ffe65cc70e65     2 days ago       845MB
sppc/dpdk-ubuntu      latest    0d5910d10e3f     2 days ago       1.66GB
<none>                <none>    d52d2f86a3c0     2 days ago       551MB
ubuntu                latest    452a96d81c30     5 weeks ago      79.6MB
```

Note: The Name of container image is defined as a set of target, name and version of Linux distoribution. For example, container image targetting dpdk apps on Ubuntu 18.04 is named as `sppc/dpdk-ubuntu:18.04`.

There are several Dockerfiles for supporting several applications and distro versions under `build/ubuntu/`. Build script understands which of Dockerfiles should be used based on the given options. If you run build script with options for dpdk and Ubuntu 18.04 as below, it finds `build/ubuntu/dpdk/Dockerfile.18.04` and runs `docker build`. Options for Linux distribution have default value, `ubuntu` and `latest`. So, you do not need to specify them if you use default.

```
# latest DPDK on latest Ubuntu
$ python3 build/main.py -t dpdk --dist-name ubuntu --dist-ver latest

# it is also the same
$ python3 build/main.py -t dpdk
```

(continues on next page)

(continued from previous page)

```
# or use Ubuntu 18.04
$ python3 build/main.py -t dpdk --dist-ver 18.04
```

Version of other than distro is also configurable by specifying a branch number via command line options.

```
$ python3 build/main.py -t dpdk --dist-ver 18.04 --dpdk-branch v19.11
$ python3 build/main.py -t pktgen --dist-ver 18.04 \
  --dpdk-branch v18.02 --pktgen-branch pktgen-3.4.9
$ python3 build/main.py -t spp --dist-ver 18.04 --dpdk-branch v19.11
```

Launch SPP and App Containers

Note: In usecase described in this chapter, SPP processes other than `spp-ctl` and CLI are containerized, but it is available to run as on host for communicating with other container applications.

Before launch containers, you should set IP address of host machine as `SPP_CTL_IP` environment variable for controller to be accessed from inside containers.

```
# Set your host IP address
$ export SPP_CTL_IP=YOUR_HOST_IPADDR
```

SPP Controller

Launch `spp-ctl` and `spp.py` to be ready before primary and secondary processes.

Note: SPP controller also provides `topo term` command for containers which shows network topology in a terminal.

However, there are a few terminals supporting this feature. `mlterm` is the most useful and easy to customize. Refer *Experimental Commands* for `topo` command.

`spp-ctl` is launched in the terminal 1.

```
# Terminal 1
$ cd /path/to/spp
$ python3 src/spp-ctl/spp-ctl
```

`spp.py` is launched in the terminal 2.

```
# Terminal 2
$ cd /path/to/spp
$ python3 src/spp.py
```

SPP Primary Container

As `SPP_CTL_IP` is activated, it is able to run `app/spp-primary.py` with options. In this case, launch `spp_primary` in background mode using one core and two physical ports in terminal 3.

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/spp-primary.py -l 0 -p 0x03
```

SPP Secondary Container

`spp_nfv` is only supported for running on container currently.

Launch `spp_nfv` in terminal 3 with options for secondary ID is `1` and core list is `1-2` for using 2nd and 3rd cores. It is also run in background mode.

```
# Terminal 3
$ python3 app/spp-nfv.py -i 1 -l 1-2
```

If it is succeeded, container is running in background. You can find it with `docker ps` command.

App Container

Launch DPDK's `testpmd` as an example of app container.

Currently, most of app containers do not support ring PMD. It means that you should create vhost PMDs from SPP controller before launching the app container.

```
# Terminal 2
spp > nfv 1; add vhost:1
spp > nfv 1; add vhost:2
```

`spp_nfv` of ID 1 running inside container creates `vhost:1` and `vhost:2`. So assign them to `testpmd` with `-d` option which is for attaching vdevs as a comma separated list of resource UIDs in SPP. `testpmd` is launched in foreground mode with `-fg` option in this case.

Note: DPDK app container tries to own ports on host which are shared with host and containers by default. It causes a confliction between SPP running on host and containers and unexpected behavior.

To avoid this situation, it is required to use `-b` or `--pci-blacklist` EAL option to exclude ports on host. PCI address of port can be inspected by using `dpdk-devbind.py -s`.

To exclude `testpmd` container tries to own physical ports, you should specify PCI addresses of the ports with `-b` or `--pci-blacklist`. You can find PCI addresses from `dpdk-devbind.py -s`.

```
# Check the status of the available devices.
dpdk-devbind --status
Network devices using DPDK-compatible driver
============================================
0000:0a:00.0 '82599ES 10-Gigabit' drv=igb_uio unused=ixgbe
0000:0a:00.1 '82599ES 10-Gigabit' drv=igb_uio unused=ixgbe

Network devices using kernel driver
===================================
...
```

In this case, you should exclude `0000:0a:00.0` and `0000:0a:00.1` with `-b` option.

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/testpmd.py -l 3-4 \
  -d vhost:1,vhost:2 \
  -fg \
  -b 0000:0a:00.0 0000:0a:00.1
 sudo docker run -it \
 ...
 -b 0000:0a:00.0 \
 -b 0000:0a:00.1 \
 ...
```

Run Applications

At the end of this getting started guide, configure network paths as described in Fig. 6.2 and start forwarding from testpmd.

Fig. 6.2: SPP and testpmd on containers

In terminal 2, add `ring:0`, connect `vhost:1` and `vhost:2` with it.

```
# Terminal 2
spp > nfv 1; add ring:0
spp > nfv 1; patch vhost:1 ring:0
spp > nfv 1; patch ring:0 vhost:2
spp > nfv 1; forward
spp > nfv 1; status
- status: running
- lcore_ids:
  - master: 1
  - slave: 2
- ports:
  - ring:0 -> vhost:2
  - vhost:1 -> ring:0
  - vhost:2
```

Start forwarding on port 0 by `start tx_first`.

```
# Terminal 3
testpmd> start tx_first
io packet forwarding - ports=2 - cores=1 - streams=2 - NUMA support...
Logical Core 4 (socket 0) forwards packets on 2 streams:
  RX P=0/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer=02:00:00:00:00:01
```

(continues on next page)

(continued from previous page)

```
  RX P=1/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer=02:00:00:00:00:00
...
```

Finally, stop forwarding to show statistics as the result. In this case, about 35 million packets are forwarded.

```
# Terminal 3
testpmd> stop
Telling cores to stop...
Waiting for lcores to finish...

  ---------------------- Forward statistics for port 0  ------------------
  RX-packets: 0              RX-dropped: 0             RX-total: 0
  TX-packets: 35077664       TX-dropped: 0             TX-total: 35077664
  ----------------------------------------------------------------------------

  ---------------------- Forward statistics for port 1  ------------------
  RX-packets: 35077632       RX-dropped: 0             RX-total: 35077632
  TX-packets: 32             TX-dropped: 0             TX-total: 32
  ----------------------------------------------------------------------------

  ++++++++++++++ Accumulated forward statistics for all ports++++++++++++
  RX-packets: 35077632       RX-dropped: 0             RX-total: 35077632
  TX-packets: 35077696       TX-dropped: 0             TX-total: 35077696
  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

6.1.3 Install

Required Packages

You need to install packages required for DPDK, and docker.

- DPDK 17.11 or later (supporting container)
- docker

Configurations

You might need some additional non-mandatory configurations.

Run docker without sudo

You should run docker as sudo in most of scripts provided in SPP container because for running DPDK applications.

However, you can run docker without sudo if you do not run DPDK applications. It is useful if you run `docker kill` for terminating containerized process, `docker rm` or `docker rmi` for cleaning resources.

```
# Terminate container from docker command
$ docker kill xxxxxx_yyyyyyy

# Remove all of containers
```

(continues on next page)

(continued from previous page)

```
$ docker rm `docker ps -aq`

# Remove all of images
$ docker rmi `docker images -aq`
```

The reason for running docker requires sudo is docker daemon binds to a unix socket instead of a TCP port. Unix socket is owned by root and other users can only access it using sudo. However, you can run if you add your account to docker group.

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

To activate it, you have to logout and re-login at once.

Network Proxy

If you are behind a proxy, you should configure proxy to build an image or running container. SPP container is supportng proxy configuration for getting it from shell environments. You confirm that `http_proxy`, `https_proxy` and `no_proxy` of environmental variables are defined.

It also required to add proxy configurations for docker daemon. Proxy for docker daemon is defined as `[Service]` entry in `/etc/systemd/system/docker.service.d/http-proxy.conf`.

```
[Service]
Environment="HTTP_PROXY=http:..." "HTTPS_PROXY=https..." "NO_PROXY=..."
```

To activate it, you should restart docker daemon.

```
$ systemctl daemon-reload
$ systemctl restart docker
```

You can confirm that environments are updated by running `docker info`.

6.1.4 Build Images

As explained in *Getting Started* section, container image is built with `build/main.py`. This script is for running `docker build` with a set of `--build-args` options for building DPDK applications.

This script supports building application from any of repositories. For example, you can build SPP hosted on your repository `https://github.com/your/spp.git` with DPDK 18.11 as following.

```
$ cd /path/to/spp/tools/sppc
$ python3 build/main.py -t spp \
  --dpdk-branch v18.11 \
  --spp-repo https://github.com/your/spp.git
```

Refer all of options running with `-h` option.

```
$ python3 build/main.py -h
usage: main.py [-h] [-t TARGET] [-ci CONTAINER_IMAGE]
               [--dist-name DIST_NAME] [--dist-ver DIST_VER]
               [--dpdk-repo DPDK_REPO] [--dpdk-branch DPDK_BRANCH]
               [--pktgen-repo PKTGEN_REPO] [--pktgen-branch PKTGEN_BRANCH]
               [--spp-repo SPP_REPO] [--spp-branch SPP_BRANCH]
               [--suricata-repo SURICATA_REPO]
               [--suricata-branch SURICATA_BRANCH]
               [--only-envsh] [--dry-run]

Docker image builder for DPDK applications

optional arguments:
  -h, --help            show this help message and exit
  -t TARGET, --target TARGET
                        Build target ('dpdk', 'pktgen', 'spp' or 'suricata')
  -ci CONTAINER_IMAGE, --container-image CONTAINER_IMAGE
                        Name of container image
  --dist-name DIST_NAME
                        Name of Linux distribution
  --dist-ver DIST_VER   Version of Linux distribution
  --dpdk-repo DPDK_REPO
                        Git URL of DPDK
  --dpdk-branch DPDK_BRANCH
                        Specific version or branch of DPDK
  --pktgen-repo PKTGEN_REPO
                        Git URL of pktgen-dpdk
  --pktgen-branch PKTGEN_BRANCH
                        Specific version or branch of pktgen-dpdk
  --spp-repo SPP_REPO   Git URL of SPP
  --spp-branch SPP_BRANCH
                        Specific version or branch of SPP
  --suricata-repo SURICATA_REPO
                        Git URL of DPDK-Suricata
  --suricata-branch SURICATA_BRANCH
                        Specific version or branch of DPDK-Suricata
  --only-envsh          Create config 'env.sh' and exit without docker build
  --dry-run             Print matrix for checking and exit without docker
                        build
```

Version Control for Images

SPP container provides version control as combination of target name, Linux distribution name and version. Built images are referred such as `sppc/dpdk-ubuntu:latest`, `sppc/spp-ubuntu:16.04` or so. `sppc` is just a prefix to indicate an image of SPP container.

Build script decides a name from given options or default values. If you run build script with only target and without distribution name and version, it uses default values `ubuntu` and `latest`.

```
# build 'sppc/dpdk-ubuntu:latest'
$ python3 build/main.py -t dpdk

# build 'sppc/spp-ubuntu:16.04'
$ python3 build/main.py -t spp --dist-ver 16.04
```

Note: SPP container does not support distributions other than Ubuntu currently. It is because SPP container has no Dockerfiles for building CentOS, Fedora or so. It will be supported in a future release.

You can build any of distributions with build script if you prepare Dockerfile by yourself. How Dockerfiles are managed is described in *Dockerfiles* section.

App container scripts also understand this naming rule. For launching `testpmd` on Ubuntu 18.04, simply give `--dist-ver` to indicate the version and other options for `testpmd` itself.

```
# launch testpmd on 'sppc/dpdk-ubuntu:18.04'
$ python3 app/testpmd.py --dist-ver 18.04 -l 3-4 ...
```

But, how can we build images for different versions of DPDK, such as 18.11 and 19.11, on the same distribution? In this case, you can use `--container-image` or `-ci` option for using any of names. It is also referred from app container scripts.

```
# build image with arbitrary name
$ python3 build/main.py -t dpdk -ci sppc/dpdk18.11-ubuntu:latest \
  --dpdk-branch v18.11

# launch testpmd with '-ci'
$ python3 app/testpmd.py -ci sppc/dpdk18.11-ubuntu:latest -l 3-4 ...
```

Dockerfiles

SPP container includes Dockerfiles for each of distributions and its versions. For instance, Dockerfiles for Ubuntu are found in `build/ubuntu` directory. You notice that each of Dockerfiles has its version as a part of file name. In other words, the list of Dockerfiles under the `ubuntu` directory shows all of supported versions of Ubuntu. You can not find Dockerfiles for CentOS as `build/centos` or other distributions because it is not supported currently. It is included in a future release.

```
$ tree build/ubuntu/
build/ubuntu/
|--- dpdk
|    |--- Dockerfile.16.04
|    |--- Dockerfile.18.04
|    ---- Dockerfile.latest
|--- pktgen
|    |--- Dockerfile.16.04
|    |--- Dockerfile.18.04
|    ---- Dockerfile.latest
|--- spp
|    |--- Dockerfile.16.04
|    |--- Dockerfile.18.04
|    ---- Dockerfile.latest
---- suricata
     |--- Dockerfile.16.04
     |--- Dockerfile.18.04
     ---- Dockerfile.latest
```

Build suricata image

Building DPDK, pktgen and SPP is completed by just running `build/main.py` script. However, building suricata requires few additional few steps.

First, build an image with `main.py` script as similar to other apps. In this example, use DPDK v18.11 and Ubuntu 18.04.

```
$ python3 build/main.py -t suricata --dpdk-branch v18.11 --dist-ver 18.04
```

After build is completed, you can find image named as `sppc/suricata-ubuntu:18.04` from `docker images`. Run bash command with this image, and execute an installer script in home directory which is created while building.

```
$ docker run -it sppc/suricata-ubuntu:18.04 /bin/bash
# ./install_suricata.sh
```

It clones and compiles suricata under home directory. You can find `$HOME/DPDK_SURICATA-4_1_1` after runing this script is completed.

Although now you are ready to use suricata, it takes a little time for doing this task everytime you run the app container. For skipping this task, you can create another image from running container with `docker commit` command.

Logout and create a new docker image with `docker commit` image's container ID. In this example, new image is named as *sppc/suricata-ubuntu2:18.04*.

```
# exit
$ docker ps -a
CONTAINER_ID  sppc/suricata-ubuntu:18.04  "/bin/bash"  3 minutes ...
$ docker commit CONTAINER_ID sppc/suricata-ubuntu2:18.04
```

You can run compiled suricata with the new image with docker as following, or app container launcher with specific options as described in. *Suricata Container*.

```
$ docker run -it sppc/suricata-ubuntu:18.04 /bin/bash
# suricata --build-info
```

Inspect Inside of Container

Container is useful, but just bit annoying to inspect inside the container because it is cleaned up immediately after process is finished and there is no clue what is happened in.

`build/run.sh` is a helper script to inspect inside the container. You can run `bash` on the container to confirm behaviour of targetting application, or run any of command.

This script refers `ubuntu/dpdk/env.sh` for Ubuntu image to include environment variables. So, it is failed to `build/run.sh` if this config file does not exist. You can create it from `build/main.py` with `--only-envsh` option if you removed it accidentally.

6.1.5 App Container Launchers

App container launcher is a python script for running SPP or DPDK application on a container. As described by name, for instance, `pktgen.py` launches pktgen-dpdk inside a container.

```
$ tree app/
app/
...
|--- helloworld.py
|--- l2fwd.py
|--- l3fwd.py
|--- l3fwd-acl.py
```

(continues on next page)

(continued from previous page)

```
|--- load-balancer.py
|--- pktgen.py
|--- spp-nfv.py
|--- spp-primary.py
|--- suricata.py
---- testpmd.py
```

Setup

You should define `SPP_CTL_IP` environment variable to SPP controller be accessed from other SPP processes inside containers. SPP controller is a CLI tool for accepting user's commands.

You cannot use `127.0.0.1` or `localhost` for `SPP_CTL_IP` because SPP processes try to find SPP controller inside each of containers and fail to. From inside of the container, SPP processes should be known IP address other than `127.0.0.1` or `localhost` of host on which SPP controller running.

SPP controller should be launched before other SPP processes.

```
$ cd /path/to/spp
$ python3 src/spp.py
```

SPP Primary Container

SPP primary process is launched from `app/spp-primary.py` as an app container. It manages resources on host from inside the container. `app/spp-primary.py` calls `docker run` with `-v` option to mount hugepages and other devices in the container to share them between host and containers.

SPP primary process is launched as foreground or background mode. You can show statistics of packet forwarding if you launch it with two cores and in foreground mode. In this case, SPP primary uses one for resource management and another one for showing statistics. If you need to minimize the usage of cores, or are not interested in the statistics, you should give just one core and run in background mode. If you run SPP primary in foreground mode with one core, it shows log messages which is also referred in syslog.

Here is an example for launching SPP primary with core list 0-1 in foreground mode. You should give portmask opiton `-p` because SPP primary requires at least one port, or failed to launch. This example is assumed that host machine has two or more physical ports.

```
$ cd /path/to/spp/tools/sppc
$ python3 app/spp-primary -l 0-1 -p 0x03 -fg
```

It is another example with one core and two ports in background mode.

```
$ python3 app/spp-primary -l 0 -p 0x03
```

SPP primary is able to run with virtual devices instead of physical NICs for a case you do not have dedicated NICs for DPDK.

```
$ python3 app/spp-primary -l 0 -d vhost:1,vhost:2 -p 0x03
```

If you need to inspect a docker command without launching a container, use `--dry-run` option. It composes docker command and just display it without running the docker command.

You refer all of options with `-h` option. Options of app container scripts are categorized four types. First one is EAL option, for example `-l`, `-c` or `-m`. Second one is app container option which is a common set of options for app containers connected with SPP. So, containers of SPP processes do not have app container option. Third one is application specific option. In this case, `-n`, `-p` or `-ip`. Final one is container option, for example `--dist-name` or `--ci`. EAL options and container options are common for all of app container scripts. On the other hand, application specific options are different each other.

```
$ python3 app/spp-primary.py -h
usage: spp-primary.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                      [--vdev [VDEV [VDEV ...]]] [--socket-mem SOCKET_MEM]
                      [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                      [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                      [--single-file-segments] [--nof-memchan NOF_MEMCHAN]
                      [-d DEV_UIDS] [-v [VOLUME [VOLUME ...]]]
                      [-nq NOF_QUEUES] [--no-privileged] [-n NOF_RING]
                      [-p PORT_MASK] [-ip CTL_IP] [--ctl-port CTL_PORT]
                      [--dist-name DIST_NAME] [--dist-ver DIST_VER]
                      [--workdir WORKDIR] [--name NAME] [-ci CONTAINER_IMAGE]
                      [-fg] [--dry-run]

Launcher for spp-primary application container

optional arguments:
  -h, --help            show this help message and exit
  -l CORE_LIST, --core-list CORE_LIST
                        Core list
  -c CORE_MASK, --core-mask CORE_MASK
                        Core mask
  -m MEM, --mem MEM     Memory size (default is 1024)
  --vdev [VDEV [VDEV ...]]
                        Virtual device in the format of DPDK
  --socket-mem SOCKET_MEM
                        Memory size
  -b [PCI_BLACKLIST [PCI_BLACKLIST ...]], --pci-blacklist [PCI_BLACKLIST...
                        PCI blacklist for excluding devices
  -w [PCI_WHITELIST [PCI_WHITELIST ...]], --pci-whitelist [PCI_WHITELIST...
                        PCI whitelist for including devices
  --single-file-segments
                        Create fewer files in hugetlbfs (non-legacy mode
                        only).
  --nof-memchan NOF_MEMCHAN
                        Number of memory channels (default is 4)
  -d DEV_UIDS, --dev-uids DEV_UIDS
                        Virtual devices of SPP in resource UID format
  -v [VOLUME [VOLUME ...]], --volume [VOLUME [VOLUME ...]]
                        Bind mount a volume (for docker)
  -nq NOF_QUEUES, --nof-queues NOF_QUEUES
                        Number of queues of virtio (default is 1)
  --no-privileged       Disable docker's privileged mode if it's needed
  -n NOF_RING, --nof-ring NOF_RING
                        Maximum number of Ring PMD
  -p PORT_MASK, --port-mask PORT_MASK
                        Port mask
  -ip CTL_IP, --ctl-ip CTL_IP
                        IP address of spp-ctl
  --ctl-port CTL_PORT   Port for primary of spp-ctl
  --dist-name DIST_NAME
```

(continues on next page)

(continued from previous page)

```
                        Name of Linux distribution
  --dist-ver DIST_VER   Version of Linux distribution
  --workdir WORKDIR     Path of directory in which the command is launched
  --name NAME           Name of container
  -ci CONTAINER_IMAGE, --container-image CONTAINER_IMAGE
                        Name of container image
  -fg, --foreground     Run container as foreground mode
  --dry-run             Only print matrix, do not run, and exit
```

SPP Secondary Container

In SPP, there are three types of secondary process, `spp_nfv`, `spp_vf` or so. However, SPP container does only support `spp_nfv` currently.

`spp-nfv.py` launches `spp_nfv` as an app container and requires options for secondary ID and core list (or core mask).

```
$ cd /path/to/spp/tools/sppc
$ python3 app/spp-nfv.py -i 1 -l 2-3
```

Refer help for all of options and usges. It shows only application specific options for simplicity.

```
$ python3 app/spp-nfv.py -h
usage: spp-nfv.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                  [--vdev [VDEV [VDEV ...]]] [--socket-mem SOCKET_MEM]
                  [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                  [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                  [--single-file-segments] [--nof-memchan NOF_MEMCHAN]
                  [-d DEV_UIDS] [-v [VOLUME [VOLUME ...]]] [-nq NOF_QUEUES]
                  [--no-privileged] [-i SEC_ID] [-ip CTL_IP]
                  [--ctl-port CTL_PORT] [--dist-name DIST_NAME]
                  [--dist-ver DIST_VER] [--workdir WORKDIR] [--name NAME]
                  [-ci CONTAINER_IMAGE] [-fg] [--dry-run]

Launcher for spp-nfv application container

optional arguments:
  ...
  -i SEC_ID, --sec-id SEC_ID
                        Secondary ID
  -ip CTL_IP, --ctl-ip CTL_IP
                        IP address of spp-ctl
  --ctl-port CTL_PORT   Port for secondary of spp-ctl
  ...
```

L2fwd Container

`app/l2fwd.py` is a launcher script for DPDK `l2fwd` sample application. It launches `l2fwd` on a container with specified vhost interfaces.

This is an example for launching with two cores (6-7th cores) with `-l` and two vhost interfaces with `-d`. `l2fwd` requires `--port-mask` or `-p` option and the number of ports should be even number.

```
$ cd /path/to/spp/tools/sppc
$ python3 app/l2fwd.py -l 6-7 -d vhost:1,vhost:2 -p 0x03 -fg
...
```

Refer help for all of options and usges. It includes app container options, for example `-d` for vhost devices and `-nq` for the number of queues of virtio, because `l2fwd` is not a SPP process. It shows options without of EAL and container for simplicity.

```
$ python3 app/l2fwd.py -h
usage: l2fwd.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                [--vdev [VDEV [VDEV ...]]] [--socket-mem SOCKET_MEM]
                [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                [--single-file-segments] [--nof-memchan NOF_MEMCHAN]
                [-d DEV_UIDS] [-v [VOLUME [VOLUME ...]]] [-nq NOF_QUEUES]
                [--no-privileged] [-p PORT_MASK] [--dist-name DIST_NAME]
                [--dist-ver DIST_VER] [--workdir WORKDIR] [--name NAME]
                [-ci CONTAINER_IMAGE] [-fg] [--dry-run]

Launcher for l2fwd application container

optional arguments:
  ...
  -d DEV_UIDS, --dev-uids DEV_UIDS
                        Virtual devices of SPP in resource UID format
  -nq NOF_QUEUES, --nof-queues NOF_QUEUES
                        Number of queues of virtio (default is 1)
  --no-privileged       Disable docker's privileged mode if it's needed
  -p PORT_MASK, --port-mask PORT_MASK
                        Port mask
  ...
```

L3fwd Container

L3fwd application is a simple example of packet processing using the DPDK. Differed from l2fwd, the forwarding decision is made based on information read from input packet. This application provides LPM (longest prefix match) or EM (exact match) methods for packet classification.

`app/l3fwd.py` launches l3fwd on a container. As similar to `l3fwd` application, this python script takes several options other than EAL for port configurations and classification methods. The mandatory options for the application are `-p` for portmask and `--config` for rx as a set of combination of `(port, queue, locre)`.

Here is an example for launching l3fwd app container with two vhost interfaces and printed log messages. There are two rx ports. `(0,0,1)` is for queue of port 0 for which lcore 1 is assigned, and `(1,0,2)` is for port 1. In this case, you should add `-nq` option because the number of both of rx and tx queues are two while the default number of virtio device is one. The number of tx queues, is two in this case, is decided to be the same value as the number of lcores. In `--vdev` option setup in the script, the number of queues is defined as `virtio_...,queues=2,...`.

```
$ cd /path/to/spp/tools/sppc
$ python3 app/l3fwd.py -l 1-2 -nq 2 -d vhost:1,vhost:2 \
  -p 0x03 --config="(0,0,1),(1,0,2)" -fg
 sudo docker run \
```

(continues on next page)

(continued from previous page)

```
 -it \
 ...
 --vdev virtio_user1,queues=2,path=/var/run/usvhost1 \
 --vdev virtio_user2,queues=2,path=/var/run/usvhost2 \
 --file-prefix spp-l3fwd-container1 \
 -- \
 -p 0x03 \
 --config "(0,0,8),(1,0,9)" \
 --parse-ptype ipv4
EAL: Detected 16 lcore(s)
EAL: Auto-detected process type: PRIMARY
EAL: Multi-process socket /var/run/.spp-l3fwd-container1_unix
EAL: Probing VFIO support...
soft parse-ptype is enabled
LPM or EM none selected, default LPM on
Initializing port 0 ... Creating queues: nb_rxq=1 nb_txq=2...
LPM: Adding route 0x01010100 / 24 (0)
LPM: Adding route 0x02010100 / 24 (1)
LPM: Adding route IPV6 / 48 (0)
LPM: Adding route IPV6 / 48 (1)
txq=8,0,0 txq=9,1,0
Initializing port 1 ... Creating queues: nb_rxq=1 nb_txq=2...

Initializing rx queues on lcore 8 ... rxq=0,0,0
Initializing rx queues on lcore 9 ... rxq=1,0,0
...
```

You can increase lcores more than the number of ports, for instance, four lcores for two ports. However, remaining 3rd and 4th lcores do nothing and require `-nq 4` for tx queues.

Default classification rule is `LPM` and the routing table is defined in `dpdk/examples/l3fwd/l3fwd_lpm.c` as below.

```
static struct ipv4_l3fwd_lpm_route ipv4_l3fwd_lpm_route_array[] = {
        {IPv4(1, 1, 1, 0), 24, 0},
        {IPv4(2, 1, 1, 0), 24, 1},
        {IPv4(3, 1, 1, 0), 24, 2},
        {IPv4(4, 1, 1, 0), 24, 3},
        {IPv4(5, 1, 1, 0), 24, 4},
        {IPv4(6, 1, 1, 0), 24, 5},
        {IPv4(7, 1, 1, 0), 24, 6},
        {IPv4(8, 1, 1, 0), 24, 7},
};
```

Refer help for all of options and usges. It shows options without of EAL and container for simplicity.

```
$ python3 app/l3fwd.py -h
usage: l3fwd.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                [--vdev [VDEV [VDEV ...]]] [--socket-mem SOCKET_MEM]
                [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                [--single-file-segments] [--nof-memchan NOF_MEMCHAN]
                [-d DEV_UIDS] [-v [VOLUME [VOLUME ...]]] [-nq NOF_QUEUES]
                [--no-privileged] [-p PORT_MASK] [--config CONFIG] [-P] [-E]
                [-L] [-dst [ETH_DEST [ETH_DEST ...]]] [--enable-jumbo]
                [--max-pkt-len MAX_PKT_LEN] [--no-numa] [--hash-entry-num]
                [--ipv6] [--parse-ptype PARSE_PTYPE] [--dist-name DIST_NAME]
                [--dist-ver DIST_VER] [--workdir WORKDIR] [--name NAME]
```

(continues on next page)

(continued from previous page)

```
                [-ci CONTAINER_IMAGE] [-fg] [--dry-run]

Launcher for l3fwd application container

optional arguments:
  ...
  -d DEV_UIDS, --dev-uids DEV_UIDS
                        Virtual devices of SPP in resource UID format
  -nq NOF_QUEUES, --nof-queues NOF_QUEUES
                        Number of queues of virtio (default is 1)
  --no-privileged       Disable docker's privileged mode if it's needed
  -p PORT_MASK, --port-mask PORT_MASK
                        (Mandatory) Port mask
  --config CONFIG       (Mandatory) Define set of port, queue, lcore for
                        ports
  -P, --promiscous      Set all ports to promiscous mode (default is None)
  -E, --exact-match     Enable exact match (default is None)
  -L, --longest-prefix-match
                        Enable longest prefix match (default is None)
  -dst [ETH_DEST [ETH_DEST ...]], --eth-dest [ETH_DEST [ETH_DEST ...]]
                        Ethernet dest for port X (X,MM:MM:MM:MM:MM:MM)
  --enable-jumbo        Enable jumbo frames, [--enable-jumbo [--max-pkt-len
                        PKTLEN]]
  --max-pkt-len MAX_PKT_LEN
                        Max packet length (64-9600) if jumbo is enabled.
  --no-numa             Disable NUMA awareness (default is None)
  --hash-entry-num      Specify the hash entry number in hexadecimal
                        (default is None)
  --ipv6                Specify the hash entry number in hexadecimal
                        (default is None)
  --parse-ptype PARSE_PTYPE
                        Set analyze packet type, ipv4 or ipv6 (default is
                        ipv4)
  ...
```

L3fwd-acl Container

L3 Forwarding with Access Control application is a simple example of packet processing using the DPDK. The application performs a security check on received packets. Packets that are in the Access Control List (ACL), which is loaded during initialization, are dropped. Others are forwarded to the correct port.

`app/l3fwd-acl.py` launches l3fwd-acl on a container. As similar to `l3fwd-acl`, this python script takes several options other than EAL for port configurations and rules. The mandatory options for the application are `-p` for portmask and `--config` for rx as a set of combination of `(port, queue, locre)`.

Here is an example for launching l3fwd app container with two vhost interfaces and printed log messages. There are two rx ports. `(0,0,1)` is for queue of port 0 for which lcore 1 is assigned, and `(1,0,2)` is for port 1. In this case, you should add `-nq` option because the number of both of rx and tx queues are two while the default number of virtio device is one. The number of tx queues, is two in this case, is decided to be the same value as the number of lcores. In `--vdev` option setup in the script, the number of queues is defined as `virtio_..,queues=2,...`.

```
$ cd /path/to/spp/tools/sppc
$ python3 app/l3fwd-acl.py -l 1-2 -nq 2 -d vhost:1,vhost:2 \
  --rule_ipv4="./rule_ipv4.db" --rule_ipv6="./rule_ipv6.db" --scalar \
  -p 0x03 --config="(0,0,1),(1,0,2)" -fg
 sudo docker run \
 -it \
 ...
 --vdev virtio_user1,queues=2,path=/var/run/usvhost1 \
 --vdev virtio_user2,queues=2,path=/var/run/usvhost2 \
 --file-prefix spp-l3fwd-container1 \
 -- \
 -p 0x03 \
 --config "(0,0,8),(1,0,9)" \
 --rule_ipv4="./rule_ipv4.db" \
 --rule_ipv6="./rule_ipv6.db" \
 --scalar
EAL: Detected 16 lcore(s)
EAL: Auto-detected process type: PRIMARY
EAL: Multi-process socket /var/run/.spp-l3fwd-container1_unix
EAL: Probing VFIO support...
soft parse-ptype is enabled
LPM or EM none selected, default LPM on
Initializing port 0 ... Creating queues: nb_rxq=1 nb_txq=2...
LPM: Adding route 0x01010100 / 24 (0)
LPM: Adding route 0x02010100 / 24 (1)
LPM: Adding route IPV6 / 48 (0)
LPM: Adding route IPV6 / 48 (1)
txq=8,0,0 txq=9,1,0
Initializing port 1 ... Creating queues: nb_rxq=1 nb_txq=2...

Initializing rx queues on lcore 8 ... rxq=0,0,0
Initializing rx queues on lcore 9 ... rxq=1,0,0
...
```

You can increase lcores more than the number of ports, for instance, four lcores for two ports. However, remaining 3rd and 4th lcores do nothing and require `-nq 4` for tx queues.

Refer help for all of options and usges. It shows options without of EAL and container for simplicity.

```
$ python3 app/l3fwd-acl.py -h
usage: l3fwd-acl.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                    [--socket-mem SOCKET_MEM]
                    [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                    [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                    [--single-file-segment] [--nof-memchan NOF_MEMCHAN]
                    [-d DEV_IDS] [-nq NOF_QUEUES] [--no-privileged]
                    [-p PORT_MASK] [--config CONFIG] [-P]
                    [--rule_ipv4 RULE_IPV4] [--rule_ipv6 RULE_IPV6]
                    [--scalar] [--enable-jumbo]
                    [--max-pkt-len MAX_PKT_LEN] [--no-numa]
                    [--dist-name DIST_NAME] [--dist-ver DIST_VER]
                    [--workdir WORKDIR] [-ci CONTAINER_IMAGE] [-fg]
                    [--dry-run]

usage: l3fwd-acl.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                    [--vdev [VDEV [VDEV ...]]] [--socket-mem SOCKET_MEM]
                    [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                    [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                    [--single-file-segments] [--nof-memchan NOF_MEMCHAN]
                    [-d DEV_UIDS] [-v [VOLUME [VOLUME ...]]]
```

(continues on next page)

(continued from previous page)

```
                      [-nq NOF_QUEUES] [--no-privileged] [-p PORT_MASK]
                      [--config CONFIG] [-P]
                      [--rule_ipv4 RULE_IPV4] [--rule_ipv6 RULE_IPV6]
                      [--scalar] [--enable-jumbo] [--max-pkt-len MAX_PKT_LEN]
                      [--no-numa] [--dist-name DIST_NAME]
                      [--dist-ver DIST_VER] [--workdir WORKDIR] [--name NAME]
                      [-ci CONTAINER_IMAGE] [-fg] [--dry-run]

Launcher for l3fwd-acl application container

optional arguments:
  ...
  -d DEV_UIDS, --dev-uids DEV_UIDS
                        Virtual devices of SPP in resource UID format
  -nq NOF_QUEUES, --nof-queues NOF_QUEUES
                        Number of queues of virtio (default is 1)
  --no-privileged       Disable docker's privileged mode if it's needed
  -p PORT_MASK, --port-mask PORT_MASK
                         (Mandatory) Port mask
  --config CONFIG        (Mandatory) Define set of port, queue, lcore for
                        ports
  -P, --promiscous      Set all ports to promiscous mode (default is None)
  --rule_ipv4 RULE_IPV4
                        Specifies the IPv4 ACL and route rules file
  --rule_ipv6 RULE_IPV6
                        Specifies the IPv6 ACL and route rules file
  --scalar              Use a scalar function to perform rule lookup
  --enable-jumbo        Enable jumbo frames, [--enable-jumbo [--max-pkt-len
                        PKTLEN]]
  --max-pkt-len MAX_PKT_LEN
                        Max packet length (64-9600) if jumbo is enabled.
  --no-numa             Disable NUMA awareness (default is None)
  ...
```

Testpmd Container

`testpmd.py` is a launcher script for DPDK's testpmd application.

It launches `testpmd` inside a container with specified vhost interfaces.

This is an example for launching with three cores (6-8th cores) and two vhost interfaces. This example is for launching `testpmd` in interactive mode.

```
$ cd /path/to/spp/tools/sppc
$ python3 app/testpmd.py -l 6-8 -d vhost:1,vhost:2 -fg -i
 sudo docker run \
 ...
 -- \
 --interactive
 ...
Checking link statuses...
Done
testpmd>
```

Testpmd has many useful options. Please refer to Running the Application section for instructions.

Note: `testpmd.py` does not support all of options of testpmd currently. You can find all

of options with `-h` option, but some of them is not implemented. If you run testpmd with not supported option, It just prints an error message to mention.

```
$ python3 app/testpmd.py -l 1,2 -d vhost:1,vhost:2 \
  --port-topology=chained
Error: '--port-topology' is not supported yet
```

Refer help for all of options and usges. It shows options without of EAL and container.

```
$ python3 app/testpmd.py -h
usage: testpmd.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                  [--vdev [VDEV [VDEV ...]]] [--socket-mem SOCKET_MEM]
                  [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                  [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                  [--single-file-segments]
                  [--nof-memchan NOF_MEMCHAN] [-d DEV_UIDS]
                  [-v [VOLUME [VOLUME ...]]]
                  [-nq NOF_QUEUES] [--no-privileged] [--pci] [-i] [-a]
                  [--tx-first] [--stats-period STATS_PERIOD]
                  [--nb-cores NB_CORES] [--coremask COREMASK]
                  [--portmask PORTMASK] [--no-numa]
                  [--port-numa-config PORT_NUMA_CONFIG]
                  [--ring-numa-config RING_NUMA_CONFIG]
                  [--socket-num SOCKET_NUM] [--mbuf-size MBUF_SIZE]
                  [--total-num-mbufs TOTAL_NUM_MBUFS]
                  [--max-pkt-len MAX_PKT_LEN]
                  [--eth-peers-configfile ETH_PEERS_CONFIGFILE]
                  [--eth-peer ETH_PEER] [--pkt-filter-mode PKT_FILTER_MODE]
                  [--pkt-filter-report-hash PKT_FILTER_REPORT_HASH]
                  [--pkt-filter-size PKT_FILTER_SIZE]
                  [--pkt-filter-flexbytes-offset PKT_FILTER_FLEXBYTES_OFFSET]
                  [--pkt-filter-drop-queue PKT_FILTER_DROP_QUEUE]
                  [--disable-crc-strip] [--enable-lro] [--enable-rx-cksum]
                  [--enable-scatter] [--enable-hw-vlan]
                  [--enable-hw-vlan-filter]
                  [--enable-hw-vlan-strip] [--enable-hw-vlan-extend]
                  [--enable-drop-en] [--disable-rss]
                  [--port-topology PORT_TOPOLOGY]
                  [--forward-mode FORWARD_MODE] [--rss-ip] [--rss-udp]
                  [--rxq RXQ] [--rxd RXD] [--txq TXQ] [--txd TXD]
                  [--burst BURST] [--mbcache MBCACHE]
                  [--rxpt RXPT] [--rxht RXHT] [--rxfreet RXFREET]
                  [--rxwt RXWT] [--txpt TXPT] [--txht TXHT] [--txwt TXWT]
                  [--txfreet TXFREET] [--txrst TXRST]
                  [--rx-queue-stats-mapping RX_QUEUE_STATS_MAPPING]
                  [--tx-queue-stats-mapping TX_QUEUE_STATS_MAPPING]
                  [--no-flush-rx] [--txpkts TXPKTS]
                  [--disable-link-check] [--no-lsc-interrupt]
                  [--no-rmv-interrupt]
                  [--bitrate-stats [BITRATE_STATS [BITRATE_STATS ...]]]
                  [--print-event PRINT_EVENT] [--mask-event MASK_EVENT]
                  [--flow-isolate-all] [--tx-offloads TX_OFFLOADS]
                  [--hot-plug] [--vxlan-gpe-port VXLAN_GPE_PORT]
                  [--mlockall] [--no-mlockall]
                  [--dist-name DIST_NAME] [--dist-ver DIST_VER]
                  [--workdir WORKDIR]
                  [--name NAME] [-ci CONTAINER_IMAGE] [-fg] [--dry-run]

Launcher for testpmd application container
```

(continues on next page)

(continued from previous page)

```
optional arguments:
  ...
  -d DEV_UIDS, --dev-uids DEV_UIDS
                        Virtual devices of SPP in resource UID format
  -nq NOF_QUEUES, --nof-queues NOF_QUEUES
                        Number of queues of virtio (default is 1)
  --no-privileged       Disable docker's privileged mode if it's needed
  --pci                 Enable PCI (default is None)
  -i, --interactive     Run in interactive mode (default is None)
  -a, --auto-start      Start forwarding on initialization (default ...)
  --tx-first            Start forwarding, after sending a burst of packets
                        first
  --stats-period STATS_PERIOD
                        Period of displaying stats, if interactive is
                        disabled
  --nb-cores NB_CORES   Number of forwarding cores
  --coremask COREMASK   Hexadecimal bitmask of the cores, do not include
                        master lcore
  --portmask PORTMASK   Hexadecimal bitmask of the ports
  --no-numa             Disable NUMA-aware allocation of RX/TX rings and RX
                        mbuf
  --port-numa-config PORT_NUMA_CONFIG
                        Specify port allocation as
                        (port,socket)[,(port,socket)]
  --ring-numa-config RING_NUMA_CONFIG
                        Specify ring allocation as
                        (port,flag,socket)[,(port,flag,socket)]
  --socket-num SOCKET_NUM
                        Socket from which all memory is allocated in NUMA
                        mode
  --mbuf-size MBUF_SIZE
                        Size of mbufs used to N (< 65536) bytes (default is
                        2048)
  --total-num-mbufs TOTAL_NUM_MBUFS
                        Number of mbufs allocated in mbuf pools, N > 1024.
  --max-pkt-len MAX_PKT_LEN
                        Maximum packet size to N (>= 64) bytes (default is
                        1518)
  --eth-peers-configfile ETH_PEERS_CONFIGFILE
                        Config file of Ether addrs of the peer ports
  --eth-peer ETH_PEER   Set MAC addr of port N as 'N,XX:XX:XX:XX:XX:XX'
  --pkt-filter-mode PKT_FILTER_MODE
                        Flow Director mode, 'none'(default), 'signature' or
                        'perfect'
  --pkt-filter-report-hash PKT_FILTER_REPORT_HASH
                        Flow Director hash match mode, 'none',
                        'match'(default) or 'always'
  --pkt-filter-size PKT_FILTER_SIZE
                        Flow Director memory size ('64K', '128K', '256K').
                        The default is 64K.
  --pkt-filter-flexbytes-offset PKT_FILTER_FLEXBYTES_OFFSET
                        Flexbytes offset (0-32, default is 0x6) defined in
                        words counted from the first byte of the dest MAC
                        address
  --pkt-filter-drop-queue PKT_FILTER_DROP_QUEUE
                        Set the drop-queue (default is 127)
  --disable-crc-strip   Disable hardware CRC stripping
  --enable-lro          Enable large receive offload
  --enable-rx-cksum     Enable hardware RX checksum offload
  --enable-scatter      Enable scatter (multi-segment) RX
  --enable-hw-vlan      Enable hardware vlan (default is None)
```

(continues on next page)

(continued from previous page)

```
  --enable-hw-vlan-filter
                        Enable hardware VLAN filter
  --enable-hw-vlan-strip
                        Enable hardware VLAN strip
  --enable-hw-vlan-extend
                        Enable hardware VLAN extend
  --enable-drop-en      Enable per-queue packet drop if no descriptors
  --disable-rss         Disable RSS (Receive Side Scaling
  --port-topology PORT_TOPOLOGY
                        Port topology, 'paired' (the default) or 'chained'
  --forward-mode FORWARD_MODE
                        Forwarding mode, 'io' (default), 'mac', 'mac_swap',
                        'flowgen', 'rxonly', 'txonly', 'csum', 'icmpecho',
                        'ieee1588', 'tm'
  --rss-ip              Set RSS functions for IPv4/IPv6 only
  --rss-udp             Set RSS functions for IPv4/IPv6 and UDP
  --rxq RXQ             Number of RX queues per port, 1-65535 (default ...)
  --rxd RXD             Number of descriptors in the RX rings
                        (default is 128)
  --txq TXQ             Number of TX queues per port, 1-65535 (default ...)
  --txd TXD             Number of descriptors in the TX rings
                        (default is 512)
  --burst BURST         Number of packets per burst, 1-512 (default is 32)
  --mbcache MBCACHE     Cache of mbuf memory pools, 0-512 (default is 16)
  --rxpt RXPT           Prefetch threshold register of RX rings
                        (default is 8)
  --rxht RXHT           Host threshold register of RX rings (default is 8)
  --rxfreet RXFREET     Free threshold of RX descriptors,0-'rxd' (...)
  --rxwt RXWT           Write-back threshold register of RX rings
                        (default is 4)
  --txpt TXPT           Prefetch threshold register of TX rings (...)
  --txht TXHT           Host threshold register of TX rings (default is 0)
  --txwt TXWT           Write-back threshold register of TX rings (...)
  --txfreet TXFREET     Free threshold of RX descriptors, 0-'txd' (...)
  --txrst TXRST         Transmit RS bit threshold of TX rings, 0-'txd'
                        (default is 0)
  --rx-queue-stats-mapping RX_QUEUE_STATS_MAPPING
                        RX queues statistics counters mapping 0-15 as
                        '(port,queue,mapping)[,(port,queue,mapping)]'
  --tx-queue-stats-mapping TX_QUEUE_STATS_MAPPING
                        TX queues statistics counters mapping 0-15 as
                        '(port,queue,mapping)[,(port,queue,mapping)]'
  --no-flush-rx         Don't flush the RX streams before starting
                        forwarding, Used mainly with the PCAP PMD
  --txpkts TXPKTS       TX segment sizes or total packet length, Valid for
                        tx-only and flowgen
  --disable-link-check  Disable check on link status when starting/stopping
                        ports
  --no-lsc-interrupt    Disable LSC interrupts for all ports
  --no-rmv-interrupt    Disable RMV interrupts for all ports
  --bitrate-stats [BITRATE_STATS [BITRATE_STATS ...]]
                        Logical core N to perform bitrate calculation
  --print-event PRINT_EVENT
                        Enable printing the occurrence of the designated
                        event, <unknown|intr_lsc|queue_state|intr_reset|
                        vf_mbox|macsec|intr_rmv|dev_probed|dev_released|
                        all>
  --mask-event MASK_EVENT
                        Disable printing the occurrence of the designated
                        event, <unknown|intr_lsc|queue_state|intr_reset|
                        vf_mbox|macsec|intr_rmv|dev_probed|dev_released|
```

(continues on next page)

(continued from previous page)

```
                        all>
  --flow-isolate-all    Providing this parameter requests flow API isolated
                        mode on all ports at initialization time
  --tx-offloads TX_OFFLOADS
                        Hexadecimal bitmask of TX queue offloads (default
                        is 0)
  --hot-plug            Enable device event monitor machenism for hotplug
  --vxlan-gpe-port VXLAN_GPE_PORT
                        UDP port number of tunnel VXLAN-GPE (default is
                        4790)
  --mlockall            Enable locking all memory
  --no-mlockall         Disable locking all memory
  ...
```

Pktgen-dpdk Container

`pktgen.py` is a launcher script for pktgen-dpdk. Pktgen is a software based traffic generator powered by the DPDK fast packet processing framework. It is not only high-performance for generating 10GB traffic with 64 byte frames, but also very configurable to handle packets with UDP, TCP, ARP, ICMP, GRE, MPLS and Queue-in-Queue. It also supports Lua for detailed configurations.

This `pktgen.py` script launches `pktgen` app container with specified vhost interfaces. Here is an example for launching with seven lcores (8-14th) and three vhost interfaces.

```
$ cd /path/to/spp/tools/sppc
$ python3 app/pktgen.py -l 8-14 -d vhost:1,vhost:2,vhost:3 \
  -fg
 sudo docker run \
 ...
 sppc/pktgen-ubuntu:latest \
 /root/dpdk/../pktgen-dpdk/app/x86_64-native-linux-gcc/pktgen \
 -l 8-14 \
 ...
 -- \
 -m [9:10].0,[11:12].1,[13:14].2
 ...
```

You notice that given lcores `-l 8-14` are assigned appropriately. Lcore 8 is used as master and remaining six lcores are use to worker threads for three ports as `-m [9:10].0, [11:12].1,[13:14].2` equally. If the number of given lcores is larger than required, remained lcores are simply not used.

Calculation of core assignment of `pktgen.py` currently is supporting up to four lcores for each of ports. If you assign fire or more lcores to a port, `pktgen.py` terminates to launch app container. It is because a usecase more than four lcores is rare and calculation is to be complicated.

```
# Assign five lcores for a slave is failed to launch
$ python3 app/pktgen.py -l 6-11 -d vhost:1
Error: Too many cores for calculation for port assignment!
Please consider to use '--matrix' for assigning directly
```

Here are other examples of lcore assignment of `pktgen.py` to help your understanding.

1. Three lcores for two ports

Assign one lcore to master and two lcores two slaves for two ports.

```
$ python3 app/pktgen.py -l 6-8 -d vhost:1,vhost:2
 ...
 -m 7.0,8.1 \
```

2. Seven lcores for three ports

Assign one lcore for master and each of two lcores to three slaves for three ports.

```
$ python3 app/pktgen.py -l 6-12 -d vhost:1,vhost:2,vhost:3
 ...
 -m [7:8].0,[9:10].1,[11:12].2 \
```

3. Seven lcores for two ports

Assign one lcore for master and each of three lcores to two slaves for two ports. In this case, each of three lcores cannot be assigned rx and tx port equally, so given two lcores to rx and one core to tx.

```
$ python3 app/pktgen.py -l 6-12 -d vhost:1,vhost:2
 ...
 -m [7-8:9].0,[10-11:12].1 \
```

Refer help for all of options and usges. It shows options without of EAL and container for simplicity.

```
$ python3 app/pktgen.py -h
usage: pktgen.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                 [--vdev [VDEV [VDEV ...]]] [--socket-mem SOCKET_MEM]
                 [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                 [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                 [--single-file-segments] [--nof-memchan NOF_MEMCHAN]
                 [-d DEV_UIDS] [-v [VOLUME [VOLUME ...]]]
                 [-nq NOF_QUEUES] [--no-privileged] [-s PCAP_FILE]
                 [-f SCRIPT_FILE]
                 [-lf LOG_FILE] [-P] [-G] [-g SOCK_ADDRESS] [-T] [-N]
                 [--matrix MATRIX] [--dist-name DIST_NAME]
                 [--dist-ver DIST_VER]
                 [--workdir WORKDIR] [--name NAME] [-ci CONTAINER_IMAGE]
                 [-fg] [--dry-run]

Launcher for pktgen-dpdk application container

optional arguments:
  ...
  -d DEV_UIDS, --dev-uids DEV_UIDS
                        Virtual devices of SPP in resource UID format
  -nq NOF_QUEUES, --nof-queues NOF_QUEUES
                        Number of queues of virtio (default is 1)
  --no-privileged       Disable docker's privileged mode if it's needed
  -s PCAP_FILE, --pcap-file PCAP_FILE
                        PCAP packet flow file of port, defined as
                        'N:filename'
  -f SCRIPT_FILE, --script-file SCRIPT_FILE
                        Pktgen script (.pkt) to or a Lua script (.lua)
  -lf LOG_FILE, --log-file LOG_FILE
                        Filename to write a log, as '-l' of pktgen
  -P, --promiscuous     Enable PROMISCUOUS mode on all ports
  -G, --sock-default    Enable socket support using default server values
                        of localhost:0x5606
```

(continues on next page)

(continued from previous page)

```
  -g SOCK_ADDRESS, --sock-address SOCK_ADDRESS
                        Same as -G but with an optional IP address and port
                        number
  -T, --term-color      Enable color terminal output in VT100
  -N, --numa            Enable NUMA support
  --matrix MATRIX       Matrix of cores and port as '-m' of pktgen, such as
                        [1:2].0 or 1.0
  ...
```

Load-Balancer Container

Load-Balancer is an application distributes packet I/O task with several worker lcores to share IP addressing.

There are three types of lcore roles in this application, rx, tx and worker lcores. Rx lcores retrieve packets from NICs and Tx lcores send it to the destinations. Worker lcores intermediate them, receive packets from rx lcores, classify by looking up the address and send it to each of destination tx lcores. Each of lcores has a set of references of lcore ID and queue as described in Application Configuration.

`load-balancer.py` expects four mandatory options.

- -rx: "(PORT, QUEUE, LCORE), . . . ", list of NIC RX ports and queues handled by the I/O RX lcores. This parameter also implicitly defines the list of I/O RX lcores.
- -tx: "(PORT, LCORE), . . . ", list of NIC TX ports handled by the I/O TX lcores. This parameter also implicitly defines the list of I/O TX lcores.
- -w: The list of the worker lcores.
- –lpm: "IP / PREFIX => PORT", list of LPM rules used by the worker lcores for packet forwarding.

Here is an example for one rx, one tx and two worker on lcores 8-10. Both of rx and rx is assinged to the same lcore 8. It receives packets from port 0 and forwards it port 0 or 1. The destination port is defined as `--lpm` option.

```
$ cd /path/to/spp/tools/sppc
$ python3 app/load-balancer.py -fg -l 8-10  -d vhost:1,vhost:2 \
  -rx "(0,0,8)" -tx "(0,8),(1,8)" -w 9,10 \
  --lpm "1.0.0.0/24=>0; 1.0.1.0/24=>1;"
```

If you are succeeded to launch the app container, it shows details of rx, tx, worker lcores and LPM rules , and starts forwarding.

```
...
Checking link statusdone
Port0 Link Up - speed 10000Mbps - full-duplex
Port1 Link Up - speed 10000Mbps - full-duplex
Initialization completed.
NIC RX ports: 0 (0 )  ;
I/O lcore 8 (socket 0): RX ports  (0, 0)  ; Output rings  0x7f9af7347...
Worker lcore 9 (socket 0) ID 0: Input rings  0x7f9af7347880  ;
Worker lcore 10 (socket 0) ID 1: Input rings  0x7f9af7345680  ;

NIC TX ports:  0  1  ;
```

(continues on next page)

(continued from previous page)

```
I/O lcore 8 (socket 0): Input rings per TX port  0 (0x7f9af7343480 ...
Worker lcore 9 (socket 0) ID 0:
Output rings per TX port  0 (0x7f9af7343480)  1 (0x7f9af7341280)  ;
Worker lcore 10 (socket 0) ID 1:
Output rings per TX port  0 (0x7f9af733f080)  1 (0x7f9af733ce80)  ;
LPM rules:
    0: 1.0.0.0/24 => 0;
    1: 1.0.1.0/24 => 1;
Ring sizes: NIC RX = 1024; Worker in = 1024; Worker out = 1024; NIC TX...
Burst sizes: I/O RX (rd = 144, wr = 144); Worker (rd = 144, wr = 144);...
Logical core 9 (worker 0) main loop.
Logical core 10 (worker 1) main loop.
Logical core 8 (I/O) main loop.
```

To stop forwarding, you need to terminate the application but might not able to with *Ctrl-C*. In this case, you can use `docker kill` command to terminate it. Find the name of container on which `load_balancer` is running and kill it.

```
$ docker ps
CONTAINER ID  IMAGE                    ...  NAMES
80ce3711b85e  sppc/dpdk-ubuntu:latest ...  competent_galileo  # kill it
281aa8f236ef  sppc/spp-ubuntu:latest  ...  youthful_mcnulty
$ docker kill competent_galileo
```

Note: You shold care about the number of worker lcores. If you add lcore 11 and assign it for third worker thread, it is failed to lauhch the application.

```
...
EAL: Probing VFIO support...
Incorrect value for --w argument (-8)

    load_balancer <EAL PARAMS> -- <APP PARAMS>

Application manadatory parameters:
    --rx "(PORT, QUEUE, LCORE), ..." : List of NIC RX ports and queues
           handled by the I/O RX lcores
...
```

The reason is the number of lcore is considered as invalid in `parse_arg_w()` as below. `n_tuples` is the number of lcores and it should be *2^n*, or returned with error code.

```
// Defined in dpdk/examples/load_balancer/config.c
static int
parse_arg_w(const char *arg)
{
        const char *p = arg;
        uint32_t n_tuples;
        ...
        if ((n_tuples & (n_tuples - 1)) != 0) {
                return -8;
        }
        ...
```

Here are other examples.

1. Separate rx and tx lcores

Use four lcores 8-11 for rx, tx and two worker threads. The number of ports is same as the previous example. You notice that rx and tx have different lcore number, 8 and 9.

```
$ python3 app/load-balancer.py -fg -l 8-11 -d vhost:1,vhost:2 \
  -rx "(0,0,8)" \
  -tx "(0,9),(1,9)" \
  -w 10,11 \
  --lpm "1.0.0.0/24=>0; 1.0.1.0/24=>1;"
```

2. Assign multiple queues for rx

To classify for three destination ports, use one rx lcore, three tx lcores and four worker lcores. In this case, rx has two queues and using `-nq 2`. You should start queue ID from 0 and to be in serial as *0,1,2,...*, or failed to launch.

```
$ python3 app/load-balancer.py -fg -l 8-13 \
  -d vhost:1,vhost:2,vhost:3 \
  -nq 2 \
  -rx "(0,0,8),(0,1,8)" \
  -tx "(0,9),(1,9),(2,9)" \
  -w 10,11,12,13 \
  --lpm "1.0.0.0/24=>0; 1.0.1.0/24=>1; 1.0.2.0/24=>2;"
```

Refer options and usages by `load-balancer.py -h`.

```
$ python3 app/load-balancer.py -h
usage: load-balancer.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                        [--vdev [VDEV [VDEV ...]]]
                        [--socket-mem SOCKET_MEM]
                        [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                        [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                        [--single-file-segments]
                        [--nof-memchan NOF_MEMCHAN]
                        [-d DEV_UIDS] [-v [VOLUME [VOLUME ...]]]
                        [-nq NOF_QUEUES] [--no-privileged]
                        [-rx RX_PORTS] [-tx TX_PORTS] [-wl WORKER_LCORES]
                        [-rsz RING_SIZES] [-bsz BURST_SIZES]
                        [--lpm LPM] [--pos-lb POS_LB]
                        [--dist-name DIST_NAME] [--dist-ver DIST_VER]
                        [--workdir WORKDIR] [--name NAME]
                        [-ci CONTAINER_IMAGE] [-fg] [--dry-run]

Launcher for load-balancer application container

optional arguments:
  ...
  -d DEV_UIDS, --dev-uids DEV_UIDS
                        Virtual devices of SPP in resource UID format
  -nq NOF_QUEUES, --nof-queues NOF_QUEUES
                        Number of queues of virtio (default is 1)
  --no-privileged       Disable docker's privileged mode if it's needed
  -rx RX_PORTS, --rx-ports RX_PORTS
                        List of rx ports and queues handled by the I/O rx
                        lcores
  -tx TX_PORTS, --tx-ports TX_PORTS
                        List of tx ports and queues handled by the I/O tx
                        lcores
  -w WORKER_LCORES, --worker-lcores WORKER_LCORES
                        List of worker lcores
  -rsz RING_SIZES, --ring-sizes RING_SIZES
                        Ring sizes of 'rx_read,rx_send,w_send,tx_written'
```

(continues on next page)

(continued from previous page)

```
  -bsz BURST_SIZES, --burst-sizes BURST_SIZES
                        Burst sizes of rx, worker or tx
  --lpm LPM             List of LPM rules
  --pos-lb POS_LB       Position of the 1-byte field used for identify
                        worker
  ...
```

Suricata Container

Suricata is a sophisticated IDS/IPS application. SPP container supports suricata 4.1.4 hosted this repository.

Unlike other scripts, `app/suricata.py` does not launch appliation directly but bash to enable to edit config file on the container. Suricata accepts options from config file specified with `--dpdk` option. You can copy your config to the container by using `docker cp`. Sample config `mysuricata.cfg` is included under `suricata-4.1.4`.

Here is an example of launching suricata with image `sppc/suricata-ubuntu2:latest` which is built as described in *Build suricata image*.

```
$ docker cp your.cnf CONTAINER_ID:/path/to/conf/your.conf
$ ./suricata.py -d vhost:1,vhost:2 -fg -ci sppc/suricata-ubuntu2:latest
# suricata --dpdk=/path/to/config
```

Refer options and usages by `load-balancer.py -h`.

```
$ python3 app/suricata.py -h
usage: suricata.py [-h] [-l CORE_LIST] [-c CORE_MASK] [-m MEM]
                   [--vdev [VDEV [VDEV ...]]] [--socket-mem SOCKET_MEM]
                   [-b [PCI_BLACKLIST [PCI_BLACKLIST ...]]]
                   [-w [PCI_WHITELIST [PCI_WHITELIST ...]]]
                   [--single-file-segments]
                   [--nof-memchan NOF_MEMCHAN] [-d DEV_UIDS]
                   [-v [VOLUME [VOLUME ...]]] [-nq NOF_QUEUES]
                   [--no-privileged]
                   [--dist-name DIST_NAME] [--dist-ver DIST_VER]
                   [--workdir WORKDIR] [--name NAME]
                   [-ci CONTAINER_IMAGE] [-fg] [--dry-run]

Launcher for suricata container

optional arguments:
  ...
  -d DEV_UIDS, --dev-uids DEV_UIDS
                        Virtual devices of SPP in resource UID format
  -nq NOF_QUEUES, --nof-queues NOF_QUEUES
                        Number of queues of virtio (default is 1)
  --no-privileged       Disable docker's privileged mode if it's needed
  --dist-name DIST_NAME
                        Name of Linux distribution
  ...
```

Helloworld Container

The helloworld sample application is an example of the simplest DPDK application that can be written.

Unlike from the other applications, it does not work as a network function actually. This app container script `helloworld.py` is intended to be used as a template for an user defined app container script. You can use it as a template for developing your app container script. An instruction for developing app container script is described in *How to Define Your App Launcher*.

Helloworld app container has no application specific options. There are only EAL and app container options. You should give `-l` option for the simplest app container.

```
$ cd /path/to/spp/tools/sppc
$ python3 app/helloworld.py -l 4-6 -fg
...
```

6.1.6 Use Cases

SPP Container provides an easy way to configure network path for DPDK application running on containers. It is useful for testing your NFV applications with `testpmd` or `pktgen` quickly, or providing a reproducible environment for evaluation with a configuration files.

In addition, using container requires less CPU and memory resources comparing with using virtual machines. It means that users can try to test variety kinds of use cases without using expensive servers.

This chapter describes examples of simple use cases of SPP container.

Perfromance Test of Vhost in Single Node

First use case is a simple performance test of vhost PMDs as shown in Fig. 6.3. Two of containers of `spp_nfv` are connected with a ring PMD and all of app container processes run on a single node.

Fig. 6.3: Test of vhost PMD in a single node

You use three terminals in this example, first one is for `spp-ctl`, second one is for SPP CLI and third one is for managing app containers. First of all, launch `spp-ctl` in terminal 1.

```
# Terminal 1
$ cd /path/to/spp
$ python3 src/spp-ctl/spp-ctl
```

Then, `spp.py` in terminal 2.

```
# Terminal 2
$ cd /path/to/spp
$ python3 src/spp.py
```

Move to terminal 3, launch app containers of `spp_primary` and `spp_nfv` step by step in background mode. You notice that vhost device is attached with `-d tap:1` which is not required if you have physical ports on host. It is because that SPP primary requires at least one port even if it is no need. You can also assign a physical port instead of this vhost device.

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/spp-primary.py -l 0 -p 0x01 -d tap:1
$ python3 app/spp-nfv.py -i 1 -l 1-2
$ python3 app/spp-nfv.py -i 2 -l 3-4
```

Then, add two vhost PMDs for pktgen app container from SPP CLI.

```
# Terminal 2
spp > nfv 1; add vhost:1
spp > nfv 2; add vhost:2
```

It is ready for launching pktgen app container. In this usecase, use five lcores for pktgen. One lcore is used for master, and remaining lcores are used for rx and tx evenly. Device ID option `-d vhost:1,vhost:2` is for refferring vhost 1 and 2.

```
# Terminal 3
$ python3 app/pktgen.py -fg -l 5-9 -d vhost:1,vhost:2
```

Finally, configure network path from SPP controller,

```
# Terminal 2
spp > nfv 1; patch ring:0 vhost:1
spp > nfv 2; patch vhost:2 ring:0
spp > nfv 1; forward
spp > nfv 2; forward
```

and start forwarding from pktgen.

```
# Terminal 3
$ Pktgen:/> start 1
```

You find that packet count of rx of port 0 and tx of port 1 is increased rapidlly.

Performance Test of Ring

Ring PMD is a very fast path to communicate between DPDK processes. It is a kind of zero-copy data passing via shared memory and better performance than vhost PMD. Currently, only `spp_nfv` provides ring PMD in SPP container. It is also possible other DPDK applications to have ring PMD interface for SPP technically, but not implemented yet.

This use case is for testing performance of ring PMDs. As described in Fig. 6.4, each of app containers on which `spp_nfv` is running are connected with ring PMDs in serial.

Fig. 6.4: Test of ring PMD

You use three terminals on host 1, first one is for `spp-ctl`, second one is for `spp.py`, and third one is for `spp_nfv` app containers. Pktgen on host 2 is started forwarding after setup on host 1 is finished.

First, launch `spp-ctl` in terminal 1.

```
# Terminal 1
$ cd /path/to/spp
$ python3 src/spp-ctl/spp-ctl
```

Then, launch `spp.py` in terminal 2.

```
# Terminal 2
$ cd /path/to/spp
$ python3 src/spp.py
```

In terminal 3, launch `spp_primary` and `spp_nfv` containers in background mode. In this case, you attach physical ports to `spp_primary` with portmask option.

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/spp-primary.py -l 0 -p 0x03
$ python3 app/spp-nfv.py -i 1 -l 1-2
$ python3 app/spp-nfv.py -i 2 -l 3-4
$ python3 app/spp-nfv.py -i 3 -l 5-6
$ python3 app/spp-nfv.py -i 4 -l 7-8
```

Note: It might happen an error to input if the number of SPP process is increased. It also might get bothered to launch several SPP processes if the number is large.

You can use `tools/spp-launcher.py` to launch SPP processes at once. Here is an example for launching `spp_primary` and four `spp_nfv` processes. `-n` is for specifying the nubmer of `spp_nfv`.

```
$ python3 tools/spp-launcher.py -n 4
```

You will find that lcore assignment is the same as below. Lcore is assigned from 0 for primary, and next two lcores for the first `spp_nfv`.

```
$ python3 app/spp-primary.py -l 0 -p 0x03
$ python3 app/spp-nfv.py -i 1 -l 1,2
$ python3 app/spp-nfv.py -i 2 -l 3,4
$ python3 app/spp-nfv.py -i 3 -l 5,6
$ python3 app/spp-nfv.py -i 4 -l 7,8
```

You can also assign lcores with `--shared` to master lcore be shared among `spp_nfv` processes. It is useful to reduce the usage of lcores as explained in *Pktgen and L2fwd using less Lcores*.

```
$ python3 tools/spp-launcher.py -n 4 --shared
```

The result of assignment of this command is the same as below. Master lcore 1 is shared among secondary processes.

```
$ python3 app/spp-primary.py -l 0 -p 0x03
$ python3 app/spp-nfv.py -i 1 -l 1,2
$ python3 app/spp-nfv.py -i 2 -l 1,3
$ python3 app/spp-nfv.py -i 3 -l 1,4
$ python3 app/spp-nfv.py -i 4 -l 1,5
```

Add ring PMDs considering which of rings is shared between which of containers. You can use recipe scripts from `playback` command instead of typing commands step by step. For this usecase example, it is included in `recipes/sppc/samples/test_ring.rcp`.

```
# Terminal 2
spp > nfv 1; add ring:0
spp > nfv 2; add ring:1
spp > nfv 2; add ring:2
spp > nfv 3; add ring:2
spp > nfv 3; add ring:3
spp > nfv 4; add ring:3
```

Then, patch all of ports to be configured containers are connected in serial.

```
# Terminal 2
spp > nfv 1; patch phy:0 ring:0
spp > nfv 2; patch ring:0 ring:1
spp > nfv 3; patch ring:1 ring:2
spp > nfv 3; patch ring:2 ring:3
spp > nfv 4; patch ring:3 phy:1
spp > nfv 1; forward
spp > nfv 2; forward
spp > nfv 3; forward
spp > nfv 4; forward
```

After setup on host 1 is finished, start forwarding from pktgen on host 2. You can see the throughput of rx and tx ports on pktgen's terminal. You also find that the throughput is almost not decreased and keeping wire rate speed even after it through several chained containers.

Pktgen and L2fwd

To consider more practical service function chaining like use case, connect not only SPP processes, but also DPDK application to `pktgen`. In this example, use `l2fwd` app container as a DPDK application for simplicity. You can also use other DPDK applications as similar to this example as described in next sections.

Fig. 6.5: Chainning pktgen and l2fwd

This configuration requires more CPUs than previous example. It is up to 14 lcores, but you can reduce lcores to do the trick. It is a trade-off between usage and performance. In this case, we focus on the usage of maximum lcores to get high performance.

Here is a list of lcore assignment for each of app containers.

- One lcore for `spp_primary` container.
- Eight lcores for four `spp_nfv` containers.
- Three lcores for `pktgen` container.
- Two lcores for `l2fwd` container.

First of all, launch `spp-ctl` and `spp.py`.

```
# Terminal 1
$ cd /path/to/spp
$ python3 src/spp-ctl/spp-ctl

# Terminal 2
$ cd /path/to/spp
$ python3 src/spp.py
```

Then, launch `spp_primary` and `spp_nfv` containers in background. It does not use physical NICs as similar to *Perfromance Test of Vhost in Single Node*. Use four of `spp_nfv` containers for using four vhost PMDs.

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/spp-primary.py -l 0 -p 0x01 -d tap:1
$ python3 app/spp-nfv.py -i 1 -l 1-2
$ python3 app/spp-nfv.py -i 2 -l 3-4
$ python3 app/spp-nfv.py -i 3 -l 5-6
$ python3 app/spp-nfv.py -i 4 -l 7-8
```

Assign ring and vhost PMDs. Each of vhost IDs to be the same as its secondary ID.

```
# Terminal 2
spp > nfv 1; add vhost:1
spp > nfv 2; add vhost:2
spp > nfv 3; add vhost:3
spp > nfv 4; add vhost:4
spp > nfv 1; add ring:0
spp > nfv 4; add ring:0
spp > nfv 2; add ring:1
spp > nfv 3; add ring:1
```

After vhost PMDs are created, you can launch containers of `pktgen` and `l2fwd`.

In this case, `pktgen` container owns vhost 1 and 2,

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/pktgen.py -l 9-11 -d vhost:1,vhost:2
```

and `l2fwd` container owns vhost 3 and 4.

```
# Terminal 4
$ cd /path/to/spp/tools/sppc
$ python app/l2fwd.py -l 12-13 -d vhost:3,vhost:4
```

Then, configure network path by pactching each of ports and start forwarding from SPP controller.

```
# Terminal 2
spp > nfv 1; patch ring:0 vhost:1
spp > nfv 2; patch vhost:2 ring:1
spp > nfv 3; patch ring:1 vhost:3
spp > nfv 4; patch vhost:4 ring:0
spp > nfv 1; forward
spp > nfv 2; forward
spp > nfv 3; forward
spp > nfv 4; forward
```

Finally, start forwarding from `pktgen` container. You can see that packet count is increased on both of `pktgen` and `l2fwd`.

For this usecase example, recipe scripts are included in `recipes/sppc/samples/pg_l2fwd.rcp`.

Pktgen and L2fwd using less Lcores

This section describes the effort of reducing the usage of lcore for *Pktgen and L2fwd*.

Here is a list of lcore assignment for each of app containers. It is totally 7 lcores while the maximum number is 14.

- One lcore for `spp_primary` container.
- Three lcores for four `spp_nfv` containers.
- Two lcores for pktgen container.
- One lcores for l2fwd container.

Fig. 6.6: Pktgen and l2fwd using less lcores

First of all, launch `spp-ctl` and `spp.py`.

```
# Terminal 1
$ cd /path/to/spp
$ python3 src/spp-ctl/spp-ctl

# Terminal 2
$ cd /path/to/spp
$ python3 src/spp.py
```

Launch `spp_primary` and `spp_nfv` containers in background. It does not use physical NICs as similar to *Perfromance Test of Vhost in Single Node*. Use two of `spp_nfv` containers for using four vhost PMDs.

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/spp-primary.py -l 0 -p 0x01 -d tap:1
$ python3 app/spp-nfv.py -i 1 -l 1,2
$ python3 app/spp-nfv.py -i 2 -l 1,3
```

The number of process and CPUs are fewer than previous example. You can reduce the number of `spp_nfv` processes by assigning several vhost PMDs to one process, although performance is decreased possibly. For the number of lcores, you can reduce it by sharing the master lcore 1 which has no heavy tasks.

Assign each of two vhost PMDs to the processes.

```
# Terminal 2
spp > nfv 1; add vhost:1
spp > nfv 1; add vhost:2
spp > nfv 2; add vhost:3
spp > nfv 2; add vhost:4
spp > nfv 1; add ring:0
spp > nfv 1; add ring:1
spp > nfv 2; add ring:0
spp > nfv 2; add ring:1
```

After vhost PMDs are created, you can launch containers of `pktgen` and `l2fwd`. These processes also share the master lcore 1 with others.

In this case, `pktgen` container uses vhost 1 and 2,

```
# Terminal 3
$ python app/pktgen.py -l 1,4,5 -d vhost:1,vhost:2
```

and `l2fwd` container uses vhost 3 and 4.

```
# Terminal 4
$ cd /path/to/spp/tools/sppc
$ python app/l2fwd.py -l 1,6 -d vhost:3,vhost:4
```

Then, configure network path by pactching each of ports and start forwarding from SPP controller.

```
# Terminal 2
spp > nfv 1; patch ring:0 vhost:1
spp > nfv 1; patch vhost:2 ring:1
spp > nfv 3; patch ring:1 vhost:3
spp > nfv 4; patch vhost:4 ring:0
spp > nfv 1; forward
spp > nfv 2; forward
spp > nfv 3; forward
spp > nfv 4; forward
```

Finally, start forwarding from `pktgen` container. You can see that packet count is increased on both of `pktgen` and `l2fwd`.

For this usecase example, recipe scripts are included in `recipes/sppc/samples/pg_l2fwd_less.rcp`.

Load-Balancer and Pktgen

Previous examples are all the single-path configurations and do not have branches. To explain how to setup a multi-path configuration, we use Load-Balancer application in this example. It is an application distributes packet I/O task with several worker lcores to share IP addressing.

Fig. 6.7: Multi-path configuration with load_balancer and pktgen

Packets from tx of `pktgen`, through ring:0, are received by rx of `load_balancer`. Then, `load_balancer` classify the packets to decide the destionations. You can count received packets on rx ports of `pktgen`.

There are six `spp_nfv` and two DPDK applications in this example. To reduce the number of lcores, configure lcore assignment to share the master lcore. Do not assign several vhosts to a process to avoid the performance degradation. It is 15 lcores required to the configuration.

Here is a list of lcore assignment for each of app containers.

- One lcore for `spp_primary` container.
- Seven lcores for four `spp_nfv` containers.
- Three lcores for `pktgen` container.
- Four lcores for `load_balancer` container.

First of all, launch `spp-ctl` and `spp.py`.

```
# Terminal 1
$ cd /path/to/spp
$ python3 src/spp-ctl/spp-ctl

# Terminal 2
$ cd /path/to/spp
$ python3 src/spp.py
```

Launch `spp_primary` and `spp_nfv` containers in background. It does not use physical NICs as similar to *Perfromance Test of Vhost in Single Node*. Use six `spp_nfv` containers for using six vhost PMDs.

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/spp-primary.py -l 0 -p 0x01 -d tap:1
$ python3 app/spp-nfv.py -i 1 -l 1,2
$ python3 app/spp-nfv.py -i 2 -l 1,3
$ python3 app/spp-nfv.py -i 3 -l 1,4
$ python3 app/spp-nfv.py -i 4 -l 1,5
$ python3 app/spp-nfv.py -i 5 -l 1,6
$ python3 app/spp-nfv.py -i 6 -l 1,7
```

Assign ring and vhost PMDs. Each of vhost IDs to be the same as its secondary ID.

```
# Terminal 2
spp > nfv 1; add vhost:1
spp > nfv 2; add vhost:2
spp > nfv 3; add vhost:3
spp > nfv 4; add vhost:4
spp > nfv 5; add vhost:5
spp > nfv 6; add vhost:6
spp > nfv 1; add ring:0
spp > nfv 2; add ring:1
spp > nfv 3; add ring:2
spp > nfv 4; add ring:0
spp > nfv 5; add ring:1
spp > nfv 6; add ring:2
```

And patch all of ports.

```
# Terminal 2
spp > nfv 1; patch vhost:1 ring:0
spp > nfv 2; patch ring:1 vhost:2
spp > nfv 3; patch ring:2 vhost:3
spp > nfv 4; patch ring:0 vhost:4
spp > nfv 5; patch vhost:5 ring:1
spp > nfv 6; patch vhost:6 ring:2
```

You had better to check that network path is configured properly. `topo` command is useful for checking it with a graphical image. Define two groups of vhost PMDs as `c1` and `c2` with `topo_subgraph` command before.

```
# Terminal 2
# define c1 and c2 to help your understanding
spp > topo_subgraph add c1 vhost:1,vhost:2,vhost:3
spp > topo_subgraph add c2 vhost:4,vhost:5,vhost:6

# show network diagram
spp > topo term
```

Finally, launch `pktgen` and `load_balancer` app containers to start traffic monitoring.

For `pktgen` container, assign lcores 8-10 and vhost 1-3. `-T` options is required to enable color terminal output.

```
# Terminal 3
$ cd /path/to/spp/tools/sppc
$ python3 app/pktgen.py -l 8-10 -d vhost:1,vhost:2,vhost:3 -T
```

For `load_balancer` container, assign lcores 12-15 and vhost 4-6. Four lcores are assigned to rx, tx and two workers. You should add `-nq` option because this example requires three or more queues. In this case, assign 4 queues.

```
# Terminal 4
$ cd /path/to/spp/tools/sppc
$ python3 app/load_balancer.py -l 11-14 \
  -d vhost:4,vhost:5,vhost:6 \
  -fg -nq 4 \
  -rx "(0,0,11),(0,1,11),(0,2,11)" \
  -tx "(0,12),(1,12),(2,12)" \
  -w 13,14 \
  --lpm "1.0.0.0/24=>0; 1.0.1.0/24=>1; 1.0.2.0/24=>2;"
```

Then, configure network path by pactching each of ports and start forwarding from SPP controller.

```
# Terminal 2
spp > nfv 1; forward
spp > nfv 2; forward
spp > nfv 3; forward
spp > nfv 4; forward
spp > nfv 5; forward
spp > nfv 6; forward
```

You start forwarding from `pktgen` container. The destination of `load_balancer` is decided by considering LPM rules. Try to classify incoming packets to port 1 on the `load_balancer` application.

On `pktgen`, change the destination IP address of port 0 to `1.0.1.100`, and start.

```
# Terminal 3
Pktgen:/> set 0 dst ip 1.0.1.100
Pktgen:/> start 0
```

As forwarding on port 0 is started, you will find the packet count of port 1 is increase rapidly. You can change the destination IP address and send packets to port 2 by stopping to forward, changing the destination IP address to `1.0.2.100` and restart forwarding.

```
# Terminal 3
Pktgen:/> stop 0
Pktgen:/> set 0 dst ip 1.0.2.100
Pktgen:/> start 0
```

You might not be able to stop `load_balancer` application with *Ctrl-C*. In this case, terminate it with `docker kill` directly as explained in *Load-Balancer Container*. You can find the name of container from `docker ps`.

For this usecase example, recipe scripts are included in `recipes/sppc/samples/lb_pg.rcp`.

6.1.7 How to Define Your App Launcher

SPP container is a set of python script for launching DPDK application on a container with docker command. You can launch your own application by preparing a container image and install your application in the container. In this chapter, you will understand how to define application container for your application.

Build Image

SPP container provides a build tool with version specific Dockerfiles. You should read the Dockerfiles to understand environmental variable or command path are defined. Build tool refer `conf/env.py` for the definitions before running docker build.

Dockerfiles of pktgen or SPP can help your understanding for building app container in which your application is placed outside of DPDK's directory. On the other hand, if you build an app container of DPDK sample application, you do not need to prepare your Dockerfile because all of examples are compiled while building DPDK's image.

Create App Container Script

As explained in *App Container Launchers*, app container script shold be prepared for each of applications. Application of SPP container is roughly categorized as DPDK sample apps or not. The former case is like that you change an existing DPDK sample application and run as a app container.

For DPDK sample apps, it is easy to build image and create app container script. On the other hand, it is a bit complex because you should you should define environmental variables, command path and compilation process by your own.

This section describes how to define app container script, first for DPDK sample applications, and then second for other than them.

DPDK Sample App Container

Procedure of App container script is defined in main() and consists of three steps of (1) parsing options, (2) setup docker command and (3) application command run inside the container.

Here is a sample code of *L2fwd Container*. `parse_args()` is defined in each of app container scripts to parse all of EAL, docker and application specific options. It returns a result of `parse_args()` method of `argparse.ArgumentParser` class. App container script uses standard library module `argparse` for parsing the arguments.

```
def main():
    args = parse_args()

    # Container image name such as 'sppc/dpdk-ubuntu:18.04'
    if args.container_image is not None:
        container_image = args.container_image
    else:
        container_image = common.container_img_name(
            common.IMG_BASE_NAMES['dpdk'],
            args.dist_name, args.dist_ver)
```

(continues on next page)

(continued from previous page)

```
    # Check for other mandatory opitons.
    if args.port_mask is None:
        common.error_exit('--port-mask')
```

If the name of container is given via `args.container_image`, it is decided as a combination of basename, distribution and its version. Basenames are defined as `IMG_BASE_NAMES` in `lib/common.py`. In general, You do not need to change for using DPDK sample apps.

```
# defined in lib/common.py
IMG_BASE_NAMES = {
    'dpdk': 'sppc/dpdk',
    'pktgen': 'sppc/pktgen',
    'spp': 'sppc/spp',
    'suricata': 'sppc/suricata',
    }
```

Options can be referred via `args`. For example, the name of container image can be referred via `args.container_image`.

Before go to step (2) and (3), you had better to check given option, expecially mandatory options. `common.error_exit()` is a helper method to print an error message for given option and do `exit()`. In this case, `--port-mask` must be given, or exit with an error message.

Setup of `sock_files` is required for creating network interfaces for the container. `sock_files()` defined in `lib/app_helper.py` is provided for creating socket files from given device UIDs.

Then, setup docker command and its options as step (2). Docker options are added by using helper method `setup_docker_opts()` which generates commonly used options for app containers. This methods returns a list of a part of options to give it to `subprocess.call()`.

```
# Setup docker command.
docker_cmd = ['sudo', 'docker', 'run', '\\']
docker_opts = app_helper.setup_docker_opts(args, sock_files)
```

You also notice that `docker_cmd` has a backslash `\\` at the end of the list. It is only used to format the printed command on the terminal. If you do no care about formatting, you do not need to add this character.

Next step is (3), to setup the application command. You should change `cmd_path` to specify your application. In `app/l2fwd.py`, the application compiled under `RTE_SDK` in DPDK's directory, but your application might be different.

```
# Setup l2fwd command run on container.
cmd_path = '{0:s}/examples/{2:s}/{1:s}/{2:s}'.format(
    env.RTE_SDK, env.RTE_TARGET, APP_NAME)

l2fwd_cmd = [cmd_path, '\\']

# Setup EAL options.
eal_opts = app_helper.setup_eal_opts(args, APP_NAME)

# Setup l2fwd options.
l2fwd_opts = ['-p', args.port_mask, '\\']
```

While setting up EAL option in `setup_eal_opts()`, `--file-prefix` is generated by using the name of application and a random number. It should be unique on the system because it

is used as the name of hugepage file.

Finally, combine command and all of options before launching from `subprocess.call()`.

```
cmds = docker_cmd + docker_opts + [container_image, '\\'] + \
    l2fwd_cmd + eal_opts + l2fwd_opts
if cmds[-1] == '\\':
    cmds.pop()
common.print_pretty_commands(cmds)

if args.dry_run is True:
    exit()

# Remove delimiters for print_pretty_commands().
while '\\' in cmds:
    cmds.remove('\\')
subprocess.call(cmds)
```

There are some optional behaviors in the final step. `common.print_pretty_commands()` replaces `\\` with a newline character and prints command line in pretty format. If you give `--dry-run` option, this launcher script prints command line and exits without launching container.

None DPDK Sample Applications in Container

There are several application using DPDK but not included in sample applications. `pktgen.py` is an example of this type of app container. As described in *DPDK Sample App Container*, app container consists of three steps and it is the same for this case.

First of all, you define parsing option for EAL, docker and your application.

```
def parse_args():
    parser = argparse.ArgumentParser(
        description="Launcher for pktgen-dpdk application container")

    parser = app_helper.add_eal_args(parser)
    parser = app_helper.add_appc_args(parser)

    parser.add_argument(
        '-s', '--pcap-file',
        type=str,
        help="PCAP packet flow file of port, defined as 'N:filename'")
    parser.add_argument(
        '-f', '--script-file',
        type=str,
        help="Pktgen script (.pkt) to or a Lua script (.lua)")
    ...

    parser = app_helper.add_sppc_args(parser)
    return parser.parse_args()
```

It is almost the same as *DPDK Sample App Container*, but it has options for `pktgen` itself. For your application, you can simply add options to `parser` object.

```
def main():
    args = parse_args()
```

Setup of socket files for network interfaces is the same as DPDK sample apps. However, you might need to change paht of command which is run in the container. In `app/pktgen.py`,

directory of `pktgen` is defined as `wd`, and the name of application s defined as `APP_NAME`. This directory can be changed with `--workdir` option.

```
# Setup docker command.
if args.workdir is not None:
    wd = args.workdir
else:
    wd = '/root/pktgen-dpdk'
docker_cmd = ['sudo', 'docker', 'run', '\\']
docker_opts = app_helper.setup_docker_opts(args, sock_files, None, wd)

# Setup pktgen command
pktgen_cmd = [APP_NAME, '\\']

# Setup EAL options.
eal_opts = app_helper.setup_eal_opts(args, APP_NAME)
```

Finally, combine all of commands and its options and launch from `subprocess.call()`.

```
cmds = docker_cmd + docker_opts + [container_image, '\\'] + \
    pktgen_cmd + eal_opts + pktgen_opts
if cmds[-1] == '\\':
    cmds.pop()
common.print_pretty_commands(cmds)

if args.dry_run is True:
    exit()

# Remove delimiters for print_pretty_commands().
while '\\' in cmds:
    cmds.remove('\\')
subprocess.call(cmds)
```

As you can see, it is almost the same as DPDK sample app container without application path and options of application specific.

6.2 Helper tools

Helper tools are intended to be used from other programs, such as `spp-ctl` or SPP CLI.

6.2.1 CPU Layout

This tool is a customized script of DPDK's user tool `cpu_layout.py`. It is used from `spp-ctl` to get CPU layout. The behaviour of this script is same as original one if you just run on terminal.

```
$ python3 tools/helpers/cpu_layout.py
======================================================================
Core and Socket Information (as reported by '/sys/devices/system/cpu')
======================================================================

cores =  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sockets =  [0]

        Socket 0
        --------
```

(continues on next page)

(continued from previous page)

```
Core 0  [0]
Core 1  [1]
...
```

Customized version of `cpu_layout.py` accepts an additional option `--json` to output the result in JSON format.

```
# Output in JSON format
$ python3 tools/helpers/cpu_layout.py --json | jq
[
  {
    "socket_id": 0,
    "cores": [
      {
        "core_id": 1,
        "cpus": [
          1
        ]
      },
      {
        "core_id": 0,
        "cpus": [
          0
        ]
      },
      ...
  }
]
```

You can almost the same result from `spp-ctl`, but the order of params are just different.

```
# Retrieve CPU layout via REST API
$ curl -X GET http://192.168.1.100:7777/v1/cpus | jq
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   505  100   505    0     0  18091      0 --:--:-- --:--:-- --:--:-- 18703
[
  {
    "cores": [
      {
        "cpus": [
          1
        ],
        "core_id": 1
      },
      {
        "cpus": [
          0
        ],
        "core_id": 0
      },
      ...
    ],
    "socket_id": 0
  }
]
```

6.2.2 Secondary Process Launcher

It is very simple python script used to lauch a secondary process from other program. It is intended to be used from spp_primary for launching. Here is whole lines of the script.

```
#!/usr/bin/env python
# coding: utf-8
"""SPP secondary launcher."""

import sys
import subprocess

if len(sys.argv) > 1:
    cmd = sys.argv[1:]
    subprocess.call(cmd)
```

As you may notice, it just runs given name or path of command with options, so you can any of command other than SPP secondary processes. However, it might be nouse for almost of users.

The reason of why this script is required is to launch secondary process from `spp_primary` indirectly to avoid launched secondaries to be zombies finally. In addtion, secondary processes other than `spp_nfv` do not work correctly after launched with execv() or other siblings directly from `spp_primary`.

6.3 Vdev_test

Vdev_test is a simple application that it forwards packets received from rx queue to tx queue on the main core. It can become a secondary process of the spp_primary. It is mainly used for testing spp_pipe but it can be used to test any virtual Ethernet devices as well.

6.3.1 Usage

```
vdev_test [EAL options] -- [--send] [--create devargs] device-name
```

Vdev_test runs foreground and stops when Ctrl-C is pressed. If `--send` option specified a packet is sent first. The virtual Ethernet device can be created to specify `--create` option.

Note: Since the device can be created by EAL `--vdev` option for a primary process, `--create` option mainly used by a secondary process.

6.3.2 Examples

Examining spp_pipe

It is assumed that pipe ports were created beforehand. First run vdev_test without `--send` option.

```
# terminal 1
$ sudo vdev_test -l 8 -n 4 --proc-type secondary -- spp_pipe0
```

Then run vdev_test with `--send` option on another terminal.

```
# terminal 2
$ sudo vdev_test -l 9 -n 4 --proc-type secondary -- --send spp_pipe1
```

Press Ctrl-C to stop processes on both terminals after for a while.

Examining vhost

This example is independent of SPP. First run vdev_test using eth_vhost0 without `--send` option.

```
# terminal 1
$ sudo vdev_test -l 8 -n 4 --vdev eht_vhost0,iface=/tmp/sock0,client=1 \
  --file-prefix=app1 -- eth_vhost0
```

Then run vdev_test using virtio_user0 with `--send` option on another terminal.

```
# terminal 1
$ sudo vdev_test -l 9 -n 4 --vdev virtio_user0,path=/tmp/sock0,server=1 \
  --file-prefix=app2 --single-file-segments -- --send virtio_user0
```

Press Ctrl-C to stop processes on both terminals after for a while.

CHAPTER 7

API Reference

7.1 spp-ctl REST API

7.1.1 Overview

`spp-ctl` provides simple REST like API. It supports http only, not https.

Request and Response

Request body is JSON format. It is accepted both `text/plain` and `application/json` for the content-type header.

A response of `GET` is JSON format and the content-type is `application/json` if the request success.

```
$ curl http://127.0.0.1:7777/v1/processes
[{"type": "primary"}, ..., {"client-id": 2, "type": "vf"}]

$ curl -X POST http://localhost:7777/v1/vfs/1/components \
  -d '{"core": 2, "name": "fwd0_tx", "type": "forward"}'
```

If a request is failed, the response is a text which shows error reason and the content-type is `text/plain`.

Error code

`spp-ctl` does basic syntax and lexical check of a request.

Table 7.1: Error codes in spp-ctl.

Error	Description
400	Syntax or lexical error, or SPP returns error for the request.
404	URL is not supported, or no SPP process of client-id in a URL.
500	System error occured in `spp-ctl`.

7.2 Independent of Process Type

7.2.1 GET /v1/processes

Show SPP processes connected with `spp-ctl`.

Response

An array of dict of processes in which process type and secondary ID are defined. So, primary process does not have this ID.

Table 7.2: Response code of getting processes.

Value	Description
200	Normal response code.

Table 7.3: Response params of getting processes.

Name	Type	Description
type	string	Process type such as `primary`, `nfv` or so.
client-id	integer	Secondary ID, so `primary` does not have it.

Examples

Request

```
$ curl -X GET -H 'application/json' \
http://127.0.0.1:7777/v1/processes
```

Response

```
[
  {
    "type": "primary"
  },
  {
    "type": "vf",
    "client-id": 1
  },
  {
```

(continues on next page)

(continued from previous page)

```
    "type": "nfv",
    "client-id": 2
  }
]
```

7.2.2 GET /v1/cpu_layout

Show CPU layout of a server on which `spp-ctl` running.

Response

An array of CPU socket params which consists of each of physical core ID and logical cores if hyper threading is enabled.

Table 7.4: Response code of CPU layout.

Value	Description
200	Normal response code.

Table 7.5: Response params of getting CPU layout.

Name	Type	Description
cores	array	Set of cores on a socket.
core_id	integer	ID of physical core.
lcores	array	Set of IDs of logical cores.
socket_id	integer	Socket ID.

Examples

Request

```
$ curl -X GET -H 'application/json' \
http://127.0.0.1:7777/v1/cpu_layout
```

Response

```
[
  {
    "cores": [
      {
        "core_id": 1,
        "lcores": [2, 3]
      },
      {
        "core_id": 0,
        "lcores": [0, 1]
      },
      {
```

(continues on next page)

(continued from previous page)

```
        "core_id": 2,
        "lcores": [4, 5]
      }
      {
        "core_id": 3,
        "lcores": [6, 7]
      }
    ],
    "socket_id": 0
  }
]
```

7.2.3 GET /v1/cpu_usage

Show CPU usage of a server on which `spp-ctl` running.

Response

An array of CPU usage of each of SPP processes. This usage consists of two params, master lcore and lcore set including master and slaves.

Table 7.6: Response code of CPU layout.

Value	Description
200	Normal response code.

Table 7.7: Response params of getting CPU layout.

Name	Type	Description
proc-type	string	Proc type such as `primary`, `nfv` or so.
master-lcore	integer	Lcore ID of master.
lcores	array	All of Lcore IDs including master and slaves.

Examples

Request

```
$ curl -X GET -H 'application/json' \
http://127.0.0.1:7777/v1/cpu_usage
```

Response

```
[
  {
    "proc-type": "primary",
    "master-lcore": 0,
    "lcores": [
      0
```

(continues on next page)

(continued from previous page)

```
    ]
  },
  {
    "proc-type": "nfv",
    "client-id": 2,
    "master-lcore": 1,
    "lcores": [1, 2]
  },
  {
    "proc-type": "vf",
    "client-id": 3,
    "master-lcore": 1,
    "lcores": [1, 3, 4, 5]
  }
]
```

7.3 spp_primary

7.3.1 GET /v1/primary/status

Show statistical information.

- Normal response codes: 200

Request example

```
$ curl -X GET -H 'application/json' \
  http://127.0.0.1:7777/v1/primary/status
```

Response

Table 7.8: Response params of primary status.

Name	Type	Description
lcores	array	Array of lcores spp_primary is using.
phy_ports	array	Array of statistics of physical ports.
ring_ports	array	Array of statistics of ring ports.
pipes	array	Array of pipe ports.

Physical port object.

Table 7.9: Attributes of physical port of primary status.

Name	Type	Description
id	integer	Port ID of the physical port.
rx	integer	The total number of received packets.
tx	integer	The total number of transferred packets.
tx_drop	integer	The total number of dropped packets of transferred.
eth	string	MAC address of the port.

Ring port object.

Table 7.10: Attributes of ring port of primary status.

Name	Type	Description
id	integer	Port ID of the ring port.
rx	integer	The total number of received packets.
rx_drop	integer	The total number of dropped packets of received.
tx	integer	The total number of transferred packets.
tx_drop	integer	The total number of dropped packets of transferred.

Pipe port object.

Table 7.11: Attributes of pipe port of primary status.

Name	Type	Description
id	integer	Port ID of the pipe port.
rx	integer	Port ID of the ring port for rx.
tx	integer	Port ID of the ring port for tx.

Response example

```
{
  "lcores": [
    0
  ],
  "phy_ports": [
    {
      "id": 0,
      "rx": 0,
      "tx": 0,
      "tx_drop": 0,
      "eth": "56:48:4f:53:54:00"
    },
    {
      "id": 1,
      "rx": 0,
      "tx": 0,
      "tx_drop": 0,
      "eth": "56:48:4f:53:54:01"
    }
  ],
  "ring_ports": [
    {
      "id": 0,
      "rx": 0,
      "rx_drop": 0,
      "tx": 0,
      "tx_drop": 0
    },
    {
      "id": 1,
      "rx": 0,
      "rx_drop": 0,
      "tx": 0,
      "tx_drop": 0
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "id": 2,
      "rx": 0,
      "rx_drop": 0,
      "tx": 0,
      "tx_drop": 0
    }
  ],
  "pipes": [
    {
      "id": 0,
      "rx": 0,
      "tx": 1
    }
  ]
}
```

7.3.2 PUT /v1/primary/forward

Start or stop forwarding.

- Normal response codes: 204
- Error response codes: 400, 404

Request example

```
$ curl -X PUT -H 'application/json' -d '{"action": "start"}' \
  http://127.0.0.1:7777/v1/primary/forward
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

Action is `start`.

```
spp > pri; forward
```

Action is `stop`.

```
spp > pri; stop
```

7.3.3 PUT /v1/primary/ports

Add or delete port.

- Normal response codes: 204
- Error response codes: 400, 404

Request (body)

Table 7.12: Request body params of ports of `spp_primary`.

Name	Type	Description
action	string	`add` or `del`.
port	string	Resource UID of {port_type}:{port_id}.
rx	string	Rx ring for pipe. It is necessary for adding pipe only.
tx	string	Tx ring for pipe. It is necessary for adding pipe only.

Request example

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "add", "port": "ring:0"}' \
  http://127.0.0.1:7777/v1/primary/ports
```

For adding pipe.

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "add", "port": "pipe:0", \
  "rx": "ring:0", "tx": "ring:1"}' \
  http://127.0.0.1:7777/v1/primary/ports
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

Not supported in SPP CLI.

7.3.4 DELETE /v1/primary/status

Clear statistical information.

- Normal response codes: 204

Request example

```
$ curl -X DELETE -H 'application/json' \
  http://127.0.0.1:7777/v1/primary/status
```

Response

There is no body content for the response of a successful `DELETE` request.

7.3.5 PUT /v1/primary/patches

Add a patch.

- Normal response codes: 204
- Error response codes: 400, 404

Request (body)

Table 7.13: Request body params of patches of `spp_primary`.

Name	Type	Description
src	string	Source port id.
dst	string	Destination port id.

Request example

```
$ curl -X PUT -H 'application/json' \
  -d '{"src": "ring:0", "dst": "ring:1"}' \
  http://127.0.0.1:7777/v1/primary/patches
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

```
spp > pri; patch {src} {dst}
```

7.3.6 DELETE /v1/primary/patches

Reset patches.

- Normal response codes: 204
- Error response codes: 400, 404

Request example

```
$ curl -X DELETE -H 'application/json' \
  http://127.0.0.1:7777/v1/primary/patches
```

Response

There is no body content for the response of a successful `DELETE` request.

Equivalent CLI command

```
spp > pri; patch reset
```

7.3.7 DELETE /v1/primary

Terminate primary process.

- Normal response codes: 204

Request example

```
$ curl -X DELETE -H 'application/json' \
  http://127.0.0.1:7777/v1/primary
```

Response

There is no body content for the response of a successful `DELETE` request.

7.3.8 PUT /v1/primary/launch

Launch a secondary process.

- Normal response codes: 204
- Error response codes: 400, 404

Request (body)

There are four params for launching secondary process. `eal` object contains a set of EAL options, and `app` contains options of teh process.

Table 7.14: Request body params for launch secondary for spp_primary.

Name	Type	Description
proc_name	string	Process name such as `spp_nfv` or `spp_vf`.
client_id	integer	Secondary ID for the process.
eal	object	Hash obj of DPDK's EAL options.
app	object	Hash obj of options of secondary process.

`eal` object is a hash of EAL options and its values. All of EAL options are referred in EAL parameters in DPDK's Getting Started Guide for Linux.

`app` object is a hash of options of secondary process, and you can refer options of each of processes in How to Use section.

Request example

Launch `spp_nfv` with secondary ID 1 and lcores `1,2`.

```
$ curl -X PUT -H 'application/json' \
  -d "{'proc_name': 'spp_nfv', 'client_id': '1', \
    'eal': {'-m': '512', '-l': '1,2', '--proc-type': 'secondary'}, \
    'app': {'-s': '192.168.11.59:6666', '-n': '1'}}"
  http://127.0.0.1:7777/v1/primary/launch
```

Launch `spp_vf` with secondary ID 2 and lcores `1,4-7`.

```
$ curl -X PUT -H 'application/json' \
  -d "{'proc_name': 'spp_vf', 'client_id': '2', \
    'eal': {'-m': '512', '-l': '1,4-7', '--proc-type': 'secondary'}, \
    'app': {'-s': '192.168.11.59:6666', '--client-id': '2'}}"
  http://127.0.0.1:7777/v1/primary/launch
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

`proc_type` is `nfv`, `vf` or `mirror` or so. `eal_opts` and `app_opts` are the same as launching from command line.

```
pri; launch {proc_type} {sec_id} {eal_opts} -- {app_opts}
```

7.3.9 POST /v1/primary/flow_rules/port_id/{port_id}/validate

Validate flow rule for specific port_id.

- Normal response codes: 200

Request example

```
$ curl -X POST \
       http://127.0.0.1:7777/v1/primary/flow_rules/port_id/0/validate \
       -H "Content-type: application/json" \
       -d '{ \
           "rule": \
              { \
              "group": 0, \
              "priority": 0, \
              "direction": "ingress", \
              "transfer": true, \
              "pattern": \
                [ \
                  "eth dst is 11:22:33:44:55:66 type mask 0xffff", \
                  "vlan vid is 100" \
                ], \
              "actions": \
```

(continues on next page)

(continued from previous page)

```
                [ \
                  "queue index 1", \
                  "of_pop_vlan" \
                ] \
              } \
            }'
```

Response

Table 7.15: Response params of validate.

Name	Type	Description
result	string	Validation result.
message	string	Additional information if any.

Response example

```
{
        "result" : "success",
        "message" : "Flow rule validated"
}
```

7.3.10 POST /v1/primary/flow_rules/port_id/{port_id}

Create flow rule for specific port_id.

- Normal response codes: 200

Request example

```
$ curl -X POST http://127.0.0.1:7777/v1/primary/flow_rules/port_id/0 \
       -H "Content-type: application/json" \
       -d '{ \
           "rule": \
              { \
              "group": 0, \
              "priority": 0, \
              "direction": "ingress", \
              "transfer": true, \
              "pattern": \
                [ \
                  "eth dst is 11:22:33:44:55:66 type mask 0xffff", \
                  "vlan vid is 100" \
                ], \
              "actions": \
                [ \
                  "queue index 1", \
                  "of_pop_vlan" \
                ] \
              } \
            }'
```

Response

Table 7.16: Response params of flow creation.

Name	Type	Description
result	string	Creation result.
message	string	Additional information if any.
rule_id	string	Rule id allocated if successful.

Response example

```
{
        "result" : "success",
        "message" : "Flow rule #0 created",
        "rule_id" : "0"
}
```

7.3.11 DELETE /v1/primary/flow_rule/port_id/{port_id}

Delete all flow rule for specific port_id.

- Normal response codes: 200

Request example

```
$ curl -X DELETE http://127.0.0.1:7777/v1/primary/flow_rule/port_id/0
```

Response

Table 7.17: Response params of flow flush.

Name	Type	Description
result	string	Deletion result.
message	string	Additional information if any.

Response example

```
{
        "result" : "success",
        "message" : "Flow rule all destroyed"
}
```

7.3.12 DELETE /v1/primary/flow_rule/{rule_id}/port_id/{port_id}

Delete specific flow rule for specific port_id.

- Normal response codes: 200

Request example

```
$ curl -X DELETE http://127.0.0.1:7777/v1/primary/flow_rules/0/port_id/0
```

Response

Table 7.18: Response params of flow deletion.

Name	Type	Description
result	string	Deletion result.
message	string	Additional information if any.

Response example

```
{
        "result" : "success",
        "message" : "Flow rule #0 destroyed"
}
```

7.4 spp_nfv

7.4.1 GET /v1/nfvs/{client_id}

Get the information of `spp_nfv`.

- Normal response codes: 200
- Error response codes: 400, 404

Request (path)

Table 7.19: Request parameter for getting info of `spp_nfv`.

Name	Type	Description
client_id	integer	client id.

Request example

```
$ curl -X GET -H 'application/json' \
  http://127.0.0.1:7777/v1/nfvs/1
```

Response

Table 7.20: Response params of getting info of `spp_nfv`.

Name	Type	Description
client-id	integer	client id.
status	string	`running` or `idling`.
ports	array	an array of port ids used by the process.
patches	array	an array of patches.

Patch ports.

Table 7.21: Attributes of patch command of `spp_nfv`.

Name	Type	Description
src	string	source port id.
dst	string	destination port id.

Response example

```
{
  "client-id": 1,
  "status": "running",
  "ports": [
    "phy:0", "phy:1", "vhost:0", "vhost:1", "ring:0", "ring:1"
  ],
  "patches": [
    {
      "src": "vhost:0", "dst": "ring:0"
    },
    {
      "src": "ring:1", "dst": "vhost:1"
    }
  ]
}
```

Equivalent CLI command

```
spp > nfv {client_id}; status
```

7.4.2 PUT /v1/nfvs/{client_id}/forward

Start or Stop forwarding.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.22: Request params of forward command of `spp_nfv`.

Name	Type	Description
client_id	integer	client id.

Request example

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "start"}' \
  http://127.0.0.1:7777/v1/nfvs/1/forward
```

Request (body)

Table 7.23: Request body params of forward of `spp_nfv`.

Name	Type	Description
action	string	`start` or `stop`.

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

Action is `start`.

```
spp > nfv {client_id}; forward
```

Action is `stop`.

```
spp > nfv {client_id}; stop
```

7.4.3 PUT /v1/nfvs/{client_id}/ports

Add or delete port.

- Normal response codes: 204
- Error response codes: 400, 404

Request(path)

Table 7.24: Request params of ports of `spp_nfv`.

Name	Type	Description
client_id	integer	client id.

Request (body)

Table 7.25: Request body params of ports of `spp_nfv`.

Name	Type	Description
action	string	`add` or `del`.
port	string	port id. port id is the form {interface_type}:{interface_id}.

Request example

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "add", "port": "ring:0"}' \
  http://127.0.0.1:7777/v1/nfvs/1/ports
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

```
spp > nfv {client_id}; {action} {if_type} {if_id}
```

7.4.4 PUT /v1/nfvs/{client_id}/patches

Add a patch.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.26: Request params of patches of `spp_nfv`.

Name	Type	Description
client_id	integer	client id.

Request (body)

Table 7.27: Request body params of patches of `spp_nfv`.

Name	Type	Description
src	string	source port id.
dst	string	destination port id.

Request example

```
$ curl -X PUT -H 'application/json' \
  -d '{"src": "ring:0", "dst": "ring:1"}' \
  http://127.0.0.1:7777/v1/nfvs/1/patches
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

```
spp > nfv {client_id}; patch {src} {dst}
```

7.4.5 DELETE /v1/nfvs/{client_id}/patches

Reset patches.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.28: Request params of deleting patches of `spp_nfv`.

Name	Type	Description
client_id	integer	client id.

Request example

```
$ curl -X DELETE -H 'application/json' \
  http://127.0.0.1:7777/v1/nfvs/1/patches
```

Response

There is no body content for the response of a successful `DELETE` request.

Equivalent CLI command

```
spp > nfv {client_id}; patch reset
```

7.4.6 DELETE /v1/nfvs/{client_id}

Terminate `spp_nfv`.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.29: Request parameter for terminating `spp_nfv`.

Name	Type	Description
client_id	integer	client id.

Request example

```
$ curl -X DELETE -H 'application/json' \
  http://127.0.0.1:7777/v1/nfvs/1
```

Response example

There is no body content for the response of a successful `DELETE` request.

Equivalent CLI command

```
spp > nfv {client_id}; exit
```

7.5 spp_vf

7.5.1 GET /v1/vfs/{client_id}

Get the information of the `spp_vf` process.

- Normal response codes: 200
- Error response codes: 400, 404

Request (path)

Table 7.30: Request parameter for getting spp_vf.

Name	Type	Description
client_id	integer	client id.

Request example

```
$ curl -X GET -H 'application/json' \
  http://127.0.0.1:7777/v1/vfs/1
```

Response

Table 7.31: Response params of getting spp_vf.

Name	Type	Description
client-id	integer	Client id.
ports	array	Array of port ids used by the process.
components	array	Array of component objects in the process.
classifier_table	array	Array of classifier tables in the process.

Component objects:

Table 7.32: Component objects of getting spp_vf.

Name	Type	Description
core	integer	Core id running on the component
name	string	Array of port ids used by the process.
type	string	Array of component objects in the process.
rx_port	array	Array of port objs connected to rx of component.
tx_port	array	Array of port objs connected to tx of component.

Port objects:

Table 7.33: Port objects of getting spp_vf.

Name	Type	Description
port	string	port id of {interface_type}:{interface_id}.
vlan	object	vlan operation which is applied to the port.

Vlan objects:

Table 7.34: Vlan objects of getting spp_vf.

Name	Type	Description
operation	string	`add`, `del` or `none`.
id	integer	vlan id.
pcp	integer	vlan pcp.

Classifier table:

Table 7.35: Vlan objects of getting spp_vf.

Name	Type	Description
type	string	`mac` or `vlan`.
value	string	mac_address or vlan_id/mac_address.
port	string	port id applied to classify.

Response example


```
{
  "client-id": 1,
  "ports": [
    "phy:0", "phy:1", "vhost:0", "vhost:1", "ring:0", "ring:1"
  ],
  "components": [
    {
      "core": 2,
      "name": "fwd0_tx",
      "type": "forward",
      "rx_port": [
        {
        "port": "ring:0",
        "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        }
      ],
      "tx_port": [
        {
          "port": "vhost:0",
          "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        }
      ]
    },
    {
      "core": 3,
      "type": "unuse"
    },
    {
      "core": 4,
      "type": "unuse"
    },
    {
      "core": 5,
      "name": "fwd1_rx",
      "type": "forward",
      "rx_port": [
        {
        "port": "vhost:1",
        "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        }
      ],
      "tx_port": [
        {
          "port": "ring:3",
          "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        }
      ]
    },
    {
      "core": 6,
      "name": "cls",
      "type": "classifier",
      "rx_port": [
        {
          "port": "phy:0",
          "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        }
      ],
      "tx_port": [
        {
```


(continues on next page)

(continued from previous page)

```
          "port": "ring:0",
          "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        },
        {
          "port": "ring:2",
          "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        }
      ]
    },
    {
      "core": 7,
      "name": "mgr1",
      "type": "merge",
      "rx_port": [
        {
          "port": "ring:1",
          "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        },
        {
          "port": "ring:3",
          "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        }
      ],
      "tx_port": [
        {
          "port": "phy:0",
          "vlan": { "operation": "none", "id": 0, "pcp": 0 }
        }
      ]
    },
  ],
  "classifier_table": [
    {
      "type": "mac",
      "value": "FA:16:3E:7D:CC:35",
      "port": "ring:0"
    }
  ]
}
```

The component which type is `unused` is to indicate unused core.

Equivalent CLI command

```
spp > vf {client_id}; status
```

7.5.2 POST /v1/vfs/{client_id}/components

Start component.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.36: Request params of components of spp_vf.

Name	Type	Description
client_id	integer	client id.

Request (body)

`type` param is oen of `forward`, `merge` or `classifier`.

Table 7.37: Response params of components of spp_vf.

Name	Type	Description
name	string	component name should be unique among processes.
core	integer	core id.
type	string	component type.

Request example

```
$ curl -X POST -H 'application/json' \
  -d '{"name": "fwd1", "core": 12, "type": "forward"}' \
  http://127.0.0.1:7777/v1/vfs/1/components
```

Response

There is no body content for the response of a successful `POST` request.

Equivalent CLI command

```
spp > vf {client_id}; component start {name} {core} {type}
```

7.5.3 DELETE /v1/vfs/{sec id}/components/{name}

Stop component.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.38: Request params of deleting component of spp_vf.

Name	Type	Description
client_id	integer	client id.
name	string	component name.

Request example

```
$ curl -X DELETE -H 'application/json' \
  http://127.0.0.1:7777/v1/vfs/1/components/fwd1
```

Response

There is no body content for the response of a successful `POST` request.

Equivalent CLI command

```
spp > vf {client_id}; component stop {name}
```

7.5.4 PUT /v1/vfs/{client_id}/components/{name}/ports

Add or delete port to the component.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.39: Request params for ports of component of spp_vf.

Name	Type	Description
client_id	integer	client id.
name	string	component name.

Request (body)

Table 7.40: Request body params for ports of component of spp_vf.

Name	Type	Description
action	string	`attach` or `detach`.
port	string	port id of {interface_type}:{interface_id}.
dir	string	`rx` or `tx`.
vlan	object	vlan operation applied to port. it can be omitted.

Vlan object:

Table 7.41: Request body params for vlan ports of component of spp_vf.

Name	Type	Description
operation	string	`add`, `del` or `none`.
id	integer	vid. ignored if operation is `del` or `none`.
pcp	integer	pcp. ignored if operation is `del` or `none`.

Request example

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "attach", "port": "vhost:1", "dir": "rx", \
       "vlan": {"operation": "add", "id": 677, "pcp": 0}}' \
  http://127.0.0.1:7777/v1/vfs/1/components/fwd1/ports
```

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "detach", "port": "vhost:0", "dir": "tx"}' \
  http://127.0.0.1:7777/v1/vfs/1/components/fwd1/ports
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

Action is `attach`.

```
spp > vf {client_id}; port add {port} {dir} {name}
```

Action is `attach` with vlan tag feature.

```
# Add vlan tag
spp > vf {client_id}; port add {port} {dir} {name} add_vlantag {id} {pcp}

# Delete vlan tag
spp > vf {client_id}; port add {port} {dir} {name} del_vlantag
```

Action is `detach`.

```
spp > vf {client_id}; port del {port} {dir} {name}
```

7.5.5 PUT /v1/vfs/{sec id}/classifier_table

Set or Unset classifier table.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.42: Request params for classifier_table of spp_vf.

Name	Type	Description
client_id	integer	client id.

Request (body)

For `vlan` param, it can be omitted if it is for `mac`.

Table 7.43: Request body params for classifier_table of spp_vf.

Name	Type	Description
action	string	`add` or `del`.
type	string	`mac` or `vlan`.
vlan	integer or null	vlan id for `vlan`. null for `mac`.
mac_address	string	mac address.
port	string	port id.

Request example

Add an entry of port `ring:0` with MAC address `FA:16:3E:7D:CC:35` to the table.

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "add", "type": "mac", \
     "mac_address": "FA:16:3E:7D:CC:35", \
     "port": "ring:0"}' \
  http://127.0.0.1:7777/v1/vfs/1/classifier_table
```

Delete an entry of port `ring:0` with MAC address `FA:16:3E:7D:CC:35` from the table.

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "del", "type": "vlan", "vlan": 475, \
     "mac_address": "FA:16:3E:7D:CC:35", "port": "ring:0"}' \
  http://127.0.0.1:7777/v1/vfs/1/classifier_table
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

Type is `mac`.

```
spp > vf {cli_id}; classifier_table {action} mac {mac_addr} {port}
```

Type is `vlan`.

```
spp > vf {cli_id}; classifier_table {action} vlan {vlan} {mac_addr} {port}
```

7.6 spp_mirror

7.6.1 GET /v1/mirrors/{client_id}

Get the information of the `spp_mirror` process.

- Normal response codes: 200
- Error response codes: 400, 404

Request (path)

Table 7.44: Request parameter for getting spp_mirror.

Name	Type	Description
client_id	integer	client id.

Request example

```
$ curl -X GET -H 'application/json' \
  http://127.0.0.1:7777/v1/mirrors/1
```

Response

Table 7.45: Response params of getting spp_mirror.

Name	Type	Description
client-id	integer	client id.
ports	array	an array of port ids used by the process.
components	array	an array of component objects in the process.

Component objects:

Table 7.46: Component objects of getting spp_mirror.

Name	Type	Description
core	integer	core id running on the component
name	string	an array of port ids used by the process.
type	string	an array of component objects in the process.
rx_port	array	an array of port objects connected to the rx side of the component.
tx_port	array	an array of port objects connected to the tx side of the component.

Port objects:

Table 7.47: Port objects of getting spp_vf.

Name	Type	Description
port	string	port id. port id is the form {interface_type}:{interface_id}.

Response example

```
{
  "client-id": 1,
  "ports": [
    "phy:0", "phy:1", "ring:0", "ring:1", "ring:2"
  ],
  "components": [
    {
      "core": 2,
      "name": "mr0",
      "type": "mirror",
      "rx_port": [
        {
        "port": "ring:0"
        }
      ],
      "tx_port": [
        {
          "port": "ring:1"
        },
        {
          "port": "ring:2"
        }
      ]
    },
    {
      "core": 3,
      "type": "unuse"
    }
  ]
}
```

The component which type is `unused` is to indicate unused core.

Equivalent CLI command

```
spp > mirror {client_id}; status
```

7.6.2 POST /v1/mirrors/{client_id}/components

Start component.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.48: Request params of components of spp_mirror.

Name	Type	Description
client_id	integer	client id.

Request (body)

Table 7.49: Response params of components of spp_mirror.

Name	Type	Description
name	string	component name. must be unique in the process.
core	integer	core id.
type	string	component type. only `mirror` is available.

Request example

```
$ curl -X POST -H 'application/json' \
  -d '{"name": "mr1", "core": 12, "type": "mirror"}' \
  http://127.0.0.1:7777/v1/mirrors/1/components
```

Response

There is no body content for the response of a successful `POST` request.

Equivalent CLI command

```
spp > mirror {client_id}; component start {name} {core} {type}
```

7.6.3 DELETE /v1/mirrors/{client_id}/components/{name}

Stop component.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.50: Request params of deleting component of spp_mirror.

Name	Type	Description
client_id	integer	client id.
name	string	component name.

Request example

```
$ curl -X DELETE -H 'application/json' \
  http://127.0.0.1:7777/v1/mirrors/1/components/mr1
```

Response

There is no body content for the response of a successful `POST` request.

Equivalent CLI command

```
spp > mirror {client_id}; component stop {name}
```

7.6.4 PUT /v1/mirrors/{client_id}/components/{name}/ports

Add or delete port to the component.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.51: Request params for ports of component of spp_mirror.

Name	Type	Description
client_id	integer	client id.
name	string	component name.

Request (body)

Table 7.52: Request body params for ports of component of spp_mirror.

Name	Type	Description
action	string	`attach` or `detach`.
port	string	port id. port id is the form {interface_type}:{interface_id}.
dir	string	`rx` or `tx`.

Request example

Attach rx port of `ring:1` to component named `mr1`.

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "attach", "port": "ring:1", "dir": "rx"}' \
  http://127.0.0.1:7777/v1/mirrors/1/components/mr1/ports
```

Detach tx port of `ring:1` from component named `mr1`.

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "detach", "port": "ring:0", "dir": "tx"}' \
  http://127.0.0.1:7777/v1/mirrors/1/components/mr1/ports
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

Action is `attach`.

```
spp > mirror {client_id}; port add {port} {dir} {name}
```

Action is `detach`.

```
spp > mirror {client_id}; port del {port} {dir} {name}
```

7.7 spp_pcap

7.7.1 GET /v1/pcaps/{client_id}

Get the information of the `spp_pcap` process.

- Normal response codes: 200
- Error response codes: 400, 404

Request (path)

Table 7.53: Request parameter for getting spp_pcap info.

Name	Type	Description
client_id	integer	client id.

Request example

```
$ curl -X GET -H 'application/json' \
  http://127.0.0.1:7777/v1/pcaps/1
```

Response

Table 7.54: Response params of getting spp_pcap.

Name	Type	Description
client-id	integer	client id.
status	string	status of the process. “running” or “idle”.
core	array	an array of core objects in the process.

Core objects:

Table 7.55: Core objects of getting spp_pcap.

Name	Type	Description
core	integer	core id
role	string	role of the task running on the core. “receive” or “write”.
rx_port	array	an array of port object for caputure. This member exists if role is “recieve”. Note that there is only a port object in the array.
filename	string	a path name of output file. This member exists if role is “write”.

There is only a port object in the array.

Port object:

Table 7.56: Port objects of getting spp_pcap.

Name	Type	Description
port	string	port id. port id is the form {interface_type}:{interface_id}.

Response example

```
{
  "client-id": 1,
  "status": "running",
  "core": [
    {
      "core": 2,
      "role": "receive",
      "rx_port": [
        {
        "port": "phy:0"
        }
      ]
    },
    {
      "core": 3,
      "role": "write",
      "filename": "/tmp/spp_pcap.20181108110600.ring0.1.2.pcap"
    }
  ]
}
```

Equivalent CLI command

```
spp > pcap {client_id}; status
```

7.7.2 PUT /v1/pcaps/{client_id}/capture

Start or Stop capturing.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.57: Request params of capture of spp_pcap.

Name	Type	Description
client_id	integer	client id.

Request (body)

Table 7.58: Request body params of capture of spp_pcap.

Name	Type	Description
action	string	`start` or `stop`.

Request example

```
$ curl -X PUT -H 'application/json' \
  -d '{"action": "start"}' \
  http://127.0.0.1:7777/v1/pcaps/1/capture
```

Response

There is no body content for the response of a successful `PUT` request.

Equivalent CLI command

Action is `start`.

```
spp > pcap {client_id}; start
```

Action is `stop`.

```
spp > pcap {client_id}; stop
```

7.7.3 DELETE /v1/pcaps/{client_id}

Terminate `spp_pcap` process.

- Normal response codes: 204
- Error response codes: 400, 404

Request (path)

Table 7.59: Request parameter for terminating spp_pcap.

Name	Type	Description
client_id	integer	client id.

Request example

```
$ curl -X DELETE -H 'application/json' \
  http://127.0.0.1:7777/v1/pcaps/1
```

Response example

There is no body content for the response of a successful `DELETE` request.

Equivalent CLI command

```
spp > pcap {client_id}; exit
```

CHAPTER 8

Bug Report

SPP is hosted project of DPDK. DPDK uses Bugzilla as its bug tracking system.

Users can issue SPP related bugs in the following link:

https://bugs.dpdk.org/enter_bug.cgi?product=SPP

Note that to issue new bug, you have to create account to the Bugzilla.

You can view open SPP related bugs in the following link:

https://bugs.dpdk.org/buglist.cgi?bug_status=__open__&product=SPP

This documentation is the latest tagged version, but some of the latest developing features might be not included. All of features not included in this documentation is described in the developing version of SPP documentation.