OvS Structure

OvS mainly have three components: kernel datapath, ovs database and ovs daemon (vswitchd).

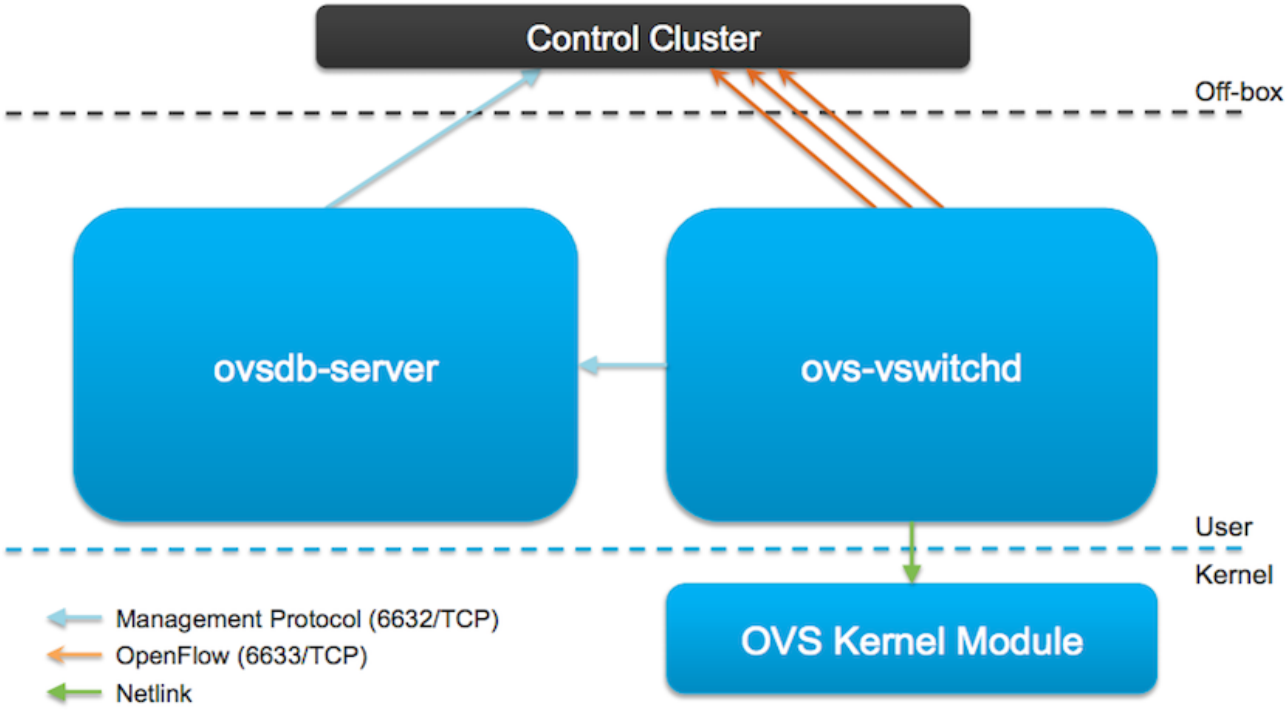

Figure 1: Open vSwitch Structure

OvS kernel datapath implement the flow matching and packet forwarding functionalites in kernel. OvS daemon implements Openflow protocol functions which are used to communicate to controller software. OvS database record virtual switches and virtual ports information. To use Openflow features and functions, the functionalities of OvS docker host in the OvS daemon component. The datapath communicate with OvS daemon via netlink events in upcall function. Netlink is an asynchronized event mechanism allows the kernel to execute a function to inform the user space applications once an kernel event is occurred.

Packet flow in OvS

Step 1: Packet arrives in OvS will be processed by a vport component, which is binded to an physical (or virtual ethernet) port in datapath.
Step 2: If the packet does not match any flow in datapath, an upcall will be send to userspace for the flow-match missing in datapath. The userspace Openflow protocol component will go through the flow entries in its record. If there is an flow match the packet, it will send the packet back to datapath and execute the actions attached on the flow. Datapath will record the flow in itself for the subsequent packets matching. Otherwise the packet will be sent to controller in an Openflow PACKET_IN message.
Step 3: If the packet match a flow in datapath, the datapath will send an upcall to userspace only contains the flow information. The userspace Openflow protocol component will compare

the flow from datapath to its flow records for any modifications on the flow, then send the updated flow back to datapath to execute actions. If the actions are different than the original records in datapath, the flow in datapath wil be updated.
Step 4: Datapath execute actions in the sequence of actions list in flow.

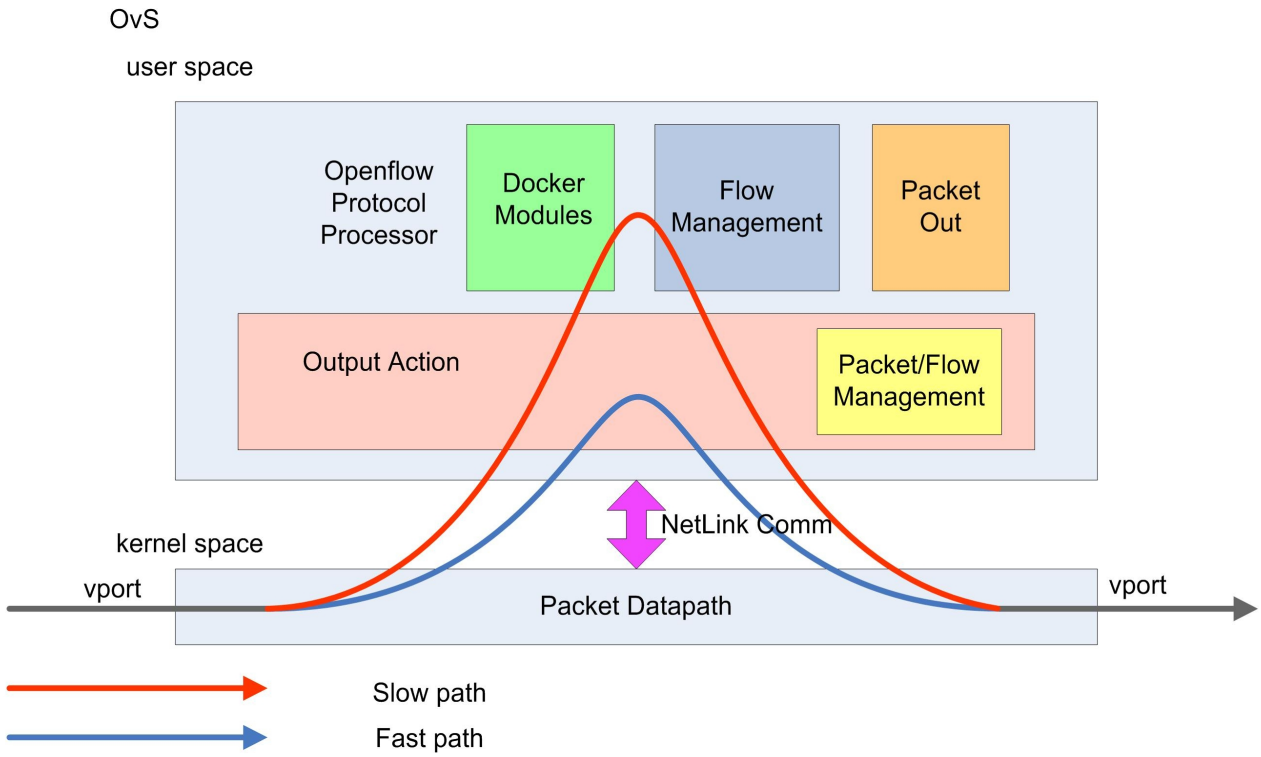

Figure 2: Packet flow in OvS

In step 2, bacause the packet missed match in any flows, the whole packet will be send to the userspace. In step 3, when packet got an match in datapath, instead of send the whole packet to userspace, only the flow in datapath will be send to userspace. Notice that the flow in datapath exactly match all the fields in flow data structure.

Whole packet could be enforced to send to userspace via upcall even when it match a flow in datapach. In this case, the packet will be marked as a "slow path" packet in its flow.

In-house controller principle

The idea of In-house controller is building an extend program component in OvS daemon, the openflow component in OvS, to process the ingress packet, execure actions and setup/remove flows in OvS daemon's flow record. In-house controller use Openflow message processing functions, like ofproto_flow_mod function and handle_packet_out function, and feed them faked message as it received from real controller to implement the controller's functionality.
The program entry is in the ofproto_init() function in ofproto/ofproto.c file. The function is named as docker_init(). It will initiate the In-House controller framework. The packet entry is in the xlate_output_action() function in ofproto/ofproto_dpif_xlate.c. In-House controller use several specific output port numbers (0xFF01-0xFFEF) as packet entry. Any actions output

packet to these ports will send packet to in-house controller. Thus, to use In-house controller, flows with generic matching rules and sprcified actions need to be set in advance to virtual switches.

In house controller in three modes:

1: Flow based packet/flow management:

In example OFPP_MAX_1(0xFF01), OFPP_MAX_2(0xFF02), OFPP_MAX_3(0xFF03), only flows was received by the userspace. In these cases, functions add_flow() and del_flow() are provided to programmers to modify flow entries in a certain virtual switch. The flow added or deleted by the current packet will not be applied to the current packet, it will only be applied to the subsequent packets. For every ingress packet in OvS, the flow will be sent to userspace and received by In-house controller, the operation on flows and packets could be variant from packet to packet.
By manipulating values in received flow, fields in current packet header could be modified. By adding multiple output actions, packet could be duplicated for multicast. Because the packet processing in this mode could be different in different packet, duplicated packets also could be changed into packets with different flows (headers).
In this mode, the packets are going through fast path in OvS, so the packet forwarding speed is kernel level speed. The limitation is the packet payload is missing. So packet deep inspection can not be apply in this mode. Another disadvantage is the program have to be placed in coordinate output action in xlate_output_action() function. It could not use modules in docker framework.

2: Packet based packet/flow management:

In example OFPP_MAX_4(0xFF04), The whole packet will be received by userspace. In this case, the packet could be deep inspected and processed in modules in docker framework. docker module functions defined in ofproto/docker_test.c file. Programmers could add new source and header files to the framework after register them in ofproto/automake.mk file. Newly added modules need to be registered in the table g_astTaskTbl in ofproto/docker_config.c. In the example, the whole packet frame was received in in xlate_output_action() function and sent to TID_TEST_RECVER module in docker. This message sending mechanism was implemented by function srv_msg_alloc()/srv_msg_send() functions. To speed up the process, these function only maintain one packet copy in memory and only send message pointer between docker modules.
In docker modules, programmers could use controller_add_flow()/controller_del_flow() functions to manage flows and actions. Programmer could also use controller_packet_out() to send out packet frames to egress ports.
The disadvantage of this mode is slow speed. Because of the slow path the packet is going through, the packet forwarding speed is 10 times slower than fast path. However, for the first packet frame was seen on each flow, these packet will also go through the slow path. So the programmer could parse the content and use docker framework eventually for the first packet.

3: Packet sniffer mode:

Packet sniffer is an module defined in docker framework, which is receiving packet frames from certain network devices. It create an vertual ethernet connection between the sniffer device and virtual switch where the flows need to be monitored. Fow each monitoring flow, it need to have an output action to the virtual connection port to the sniffer device. When packets are sent to the real destination via fast path, an copy of the packet will also be sent to sniffer device. Test result shows that this packet duplicate will not seriously impact the packet forwarding speed in original flow.
Packet sniffer module have to be registered in the table g_astTaskTbl in ofproto/docker_config.c with the name of the virtual port. The another virtual port of the connection and the monitered switch name also need to be provided to the sniffer module. Programmer have to manually add the port no to output action in the monitored flow.
In docker framework, multiple sniffer modules could be configured for one or multiple switches. Programmer could send the received packet frame to other modules in docker framework to process. Each docker module have an message queue in 500 frames size. Any frame beyond this size will be discarded. Programmer could use same interfaces in mode 2 to manage packets and flows.

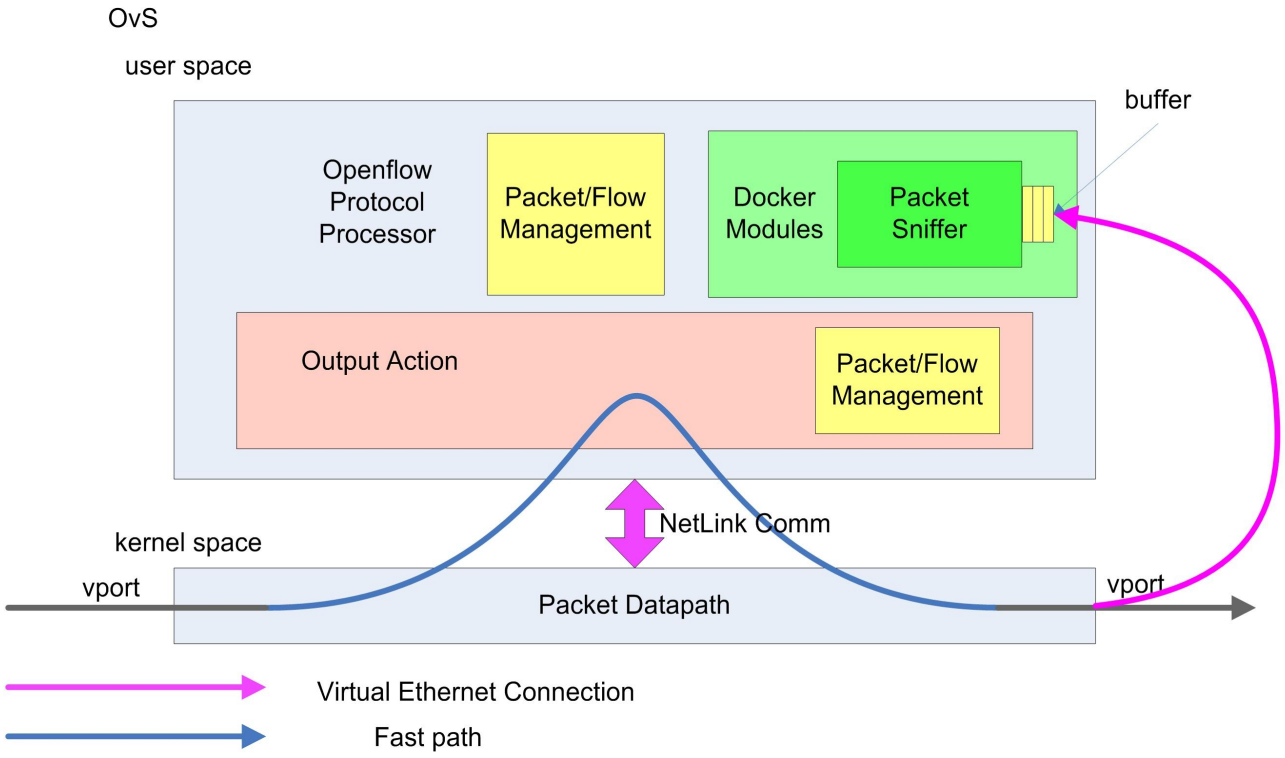

Figure 3: Sniffer mode in In-house controller

Example Applications: