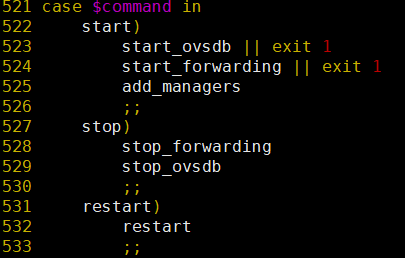
OVS分析

# ovs-vsctl start

sudo /sbin/modprobe openvswitch

sudo /usr/local/share/openvswitch/scripts/ovs-ctl start



steven@ubuntu:~/study/openvswitch-2.12.0$ sudo ./utilities/ovs-ctl start

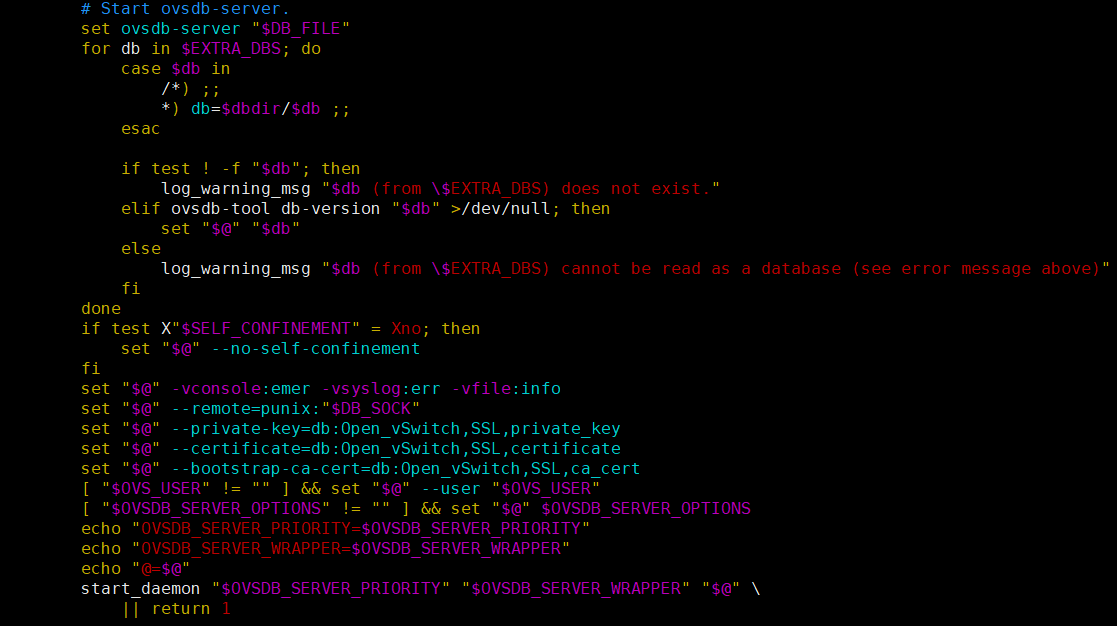
OVSDB\_SERVER=yes

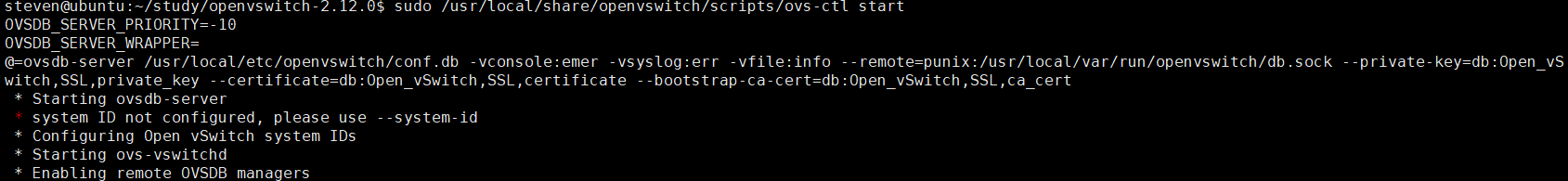
\* Starting ovsdb-server

\* system ID not configured, please use --system-id

\* Configuring Open vSwitch system IDs

./utilities/ovs-ctl: 519: ./utilities/ovs-ctl: ./utilities/ovs-kmod-ctl: not found





等价于执行了下面这句

start\_daemon -10 ovsdb-server /usr/local/etc/openvswitch/conf.db -vconsole:emer -vsyslog:err -vfile:info --remote=punix:/usr/local/var/run/openvswitch/db.sock --private-key=db:Open\_vSwitch,SSL,private\_key --certificate=db:Open\_vSwitch,SSL,certificate --bootstrap-ca-cert=db:Open\_vSwitch,SSL,ca\_cert

RFC 7047 specifies the OVSDB protocol but it does not specify an on-disk storage format.

# ovsdb创建

ovsdb-tool create /usr/local/etc/openvswitch/conf.db

/usr/local/share/openvswitch/vswitch.ovsschema

gdb --args ovsdb-tool create conf.db /usr/local/share/openvswitch/vswitch.ovsschema

# ovs-vswitchd

alias run\_ovsdb-server='sudo ovsdb-server /usr/local/etc/openvswitch/conf.db -vconsole:emer -vsyslog:err -vfile:info --remote=punix:/usr/local/var/run/openvswitch/db.sock --private-key=db:Open\_vSwitch,SSL,private\_key --certificate=db:Open\_vSwitch,SSL,certificate --bootstrap-ca-cert=db:Open\_vSwitch,SSL,ca\_cert --no-chdir --log-file=/usr/local/var/log/openvswitch/ovsdb-server.log --pidfile=/usr/local/var/run/openvswitch/ovsdb-server.pid --detach --monitor'

alias gdb\_ovs-vswitchd='sudo gdb --args /home/steven/study/openvswitch-2.12.0/vswitchd/ovs-vswitchd unix:/usr/local/var/run/openvswitch/db.sock -vconsole:emer -vsyslog:err -vfile:info --mlockall'

# ovsdb-server

alias gdb\_ovsdb-server='sudo gdb --args ovsdb-server /usr/local/etc/openvswitch/conf.db -vconsole:emer -vsyslog:err -vfile:info --remote=punix:/usr/local/var/run/openvswitch/db.sock --private-key=db:Open\_vSwitch,SSL,private\_key --certificate=db:Open\_vSwitch,SSL,certificate --bootstrap-ca-cert=db:Open\_vSwitch,SSL,ca\_cert --no-chdir --log-file=/usr/local/var/log/openvswitch/ovsdb-server.log --pidfile=/usr/local/var/run/openvswitch/ovsdb-server.pid --detach --monitor'

DEBUG 调试去掉--detach，否则主进程退出，后台运行

sudo gdb --args ovsdb-server /usr/local/etc/openvswitch/conf.db -vconsole:emer -vsyslog:err -vfile:info --remote=punix:/usr/local/var/run/openvswitch/db.sock --private-key=db:Open\_vSwitch,SSL,private\_key --certificate=db:Open\_vSwitch,SSL,certificate --bootstrap-ca-cert=db:Open\_vSwitch,SSL,ca\_cert --no-chdir --log-file=/usr/local/var/log/openvswitch/ovsdb-server.log --pidfile=/usr/local/var/run/openvswitch/ovsdb-server.pid

<https://blog.csdn.net/wuheshi/article/details/102872896>

<http://blog.chinaunix.net/uid-26133817-id-4912718.html>

1. #0  0x00007ffff7035730 in \_\_poll\_nocancel () at ../sysdeps/unix/syscall-template.S:84
2. #1  0x0000000000461851 in time\_poll (pollfds=0x6c2b50, n\_pollfds=3, handles=0x0, timeout\_when=4710702, elapsed=0x7fffffffdfb8) at lib/timeval.c:326
3. #2  0x0000000000451dd0 in poll\_block () at lib/poll-loop.c:364
4. #3  0x00000000004062cb in main\_loop (config=0x7fffffffe1f0, jsonrpc=0x6b73a0, all\_dbs=0x7fffffffe1d0, unixctl=0x6c58e0, remotes=0x7fffffffe190, run\_process=0x0, exiting=0x7fffffffe102,
5. is\_backup=0x7fffffffe101) at ovsdb/ovsdb-server.c:280
6. #4  0x0000000000406b37 in main (argc=12, argv=0x7fffffffe348) at ovsdb/ovsdb-server.c:460

作为服务器侦听端口port[:ip]  ，在客户端sudo ovsdb-client dump tcp:192.168.1.30:6646

/usr/local/sbin/ovsdb-server /usr/local/etc/openvswitch/conf.db -vconsole:emer -vsyslog:err -vfile:info --remote=ptcp:6646 --private-key=db:Open\_vSwitch,SSL,private\_key --certificate=db:Open\_vSwitch,SSL,certificate --bootstrap-ca-cert=db:Open\_vSwitch,SSL,ca\_cert --no-chdir --log-file=/usr/local/var/log/openvswitch/ovsdb-server.log --pidfile=/usr/local/var/run/openvswitch/ovsdb-server.pid

1. ptcp:port[:ip]
2. Listen  on  the  given  TCP  port  **for**  a connection.  By
3. **default**, connections are not bound to a particular  local
4. IP  address  and  it  listens only on IPv4 (but not IPv6)
5. addresses, but ip may be specified  to  listen  only  **for**
6. connections to the given ip, either IPv4 or IPv6 address.
7. If ip is an IPv6 address, then wrap ip with square brack‐
8. ets, e.g.: ptcp:6640:[::1].
9. ovsdb/ovsdb-server.c:
11. **int**
12. main(**int** argc, **char** \*argv[])
13. {
14. /\* step.1. handle configs, open db \*/
15. SSET\_FOR\_EACH (db\_filename, &db\_filenames)
16. open\_db(&server\_config, db\_filename);
18. /\* step.2. create unixctl server and register commands \*/
19. unixctl\_server\_create(unixctl\_path, &unixctl);
20. unixctl\_command\_register("exit", "", 0, 0, ovsdb\_server\_exit, &exiting);
21. ...
23. /\* step.3. enter main loop \*/
24. main\_loop();
25. |--**while** (!\*exiting) {
26. /\* step.3.1 handle control messages from CLI and RPC \*/
27. unixctl\_server\_run(unixctl);       // handle CLI commands (turn into RPC requests)
28. ovsdb\_jsonrpc\_server\_run(jsonrpc); // handle RPC requests
30. /\* step.3.2 ovsdb session execute \*/
31. SHASH\_FOR\_EACH(node, all\_dbs) {
32. ovsdb\_trigger\_run(db->db, time\_msec()); // execute session, commit changes
33. }
35. /\* step.3.3 update Manager status(es) every 2.5 seconds \*/
36. **if** (time\_msec() >= status\_timer)
37. update\_remote\_status(jsonrpc, remotes, all\_dbs);
39. /\* step.3.4 wait events arriving \*/
41. /\* step.3.5 block until events arrive \*/
42. poll\_block();
43. }
45. /\* step.4. clean and exit \*/
46. }

# datapath

## 编译

./configure CFLAGS='-g -O0'

./configure --with-linux=/lib/modules/`uname -r`/build CFLAGS='-g -O0'

./configure --with-linux=/home/steven/work/linux-4.1.30 CFLAGS='-g -O0'

编译错误

/usr/src/linux-headers-4.15.0-99/include/net$ vi dst\_ops.h

sudo /sbin/modprobe openvswitch

/sbin/lsmod | grep openvswitch

sudo /sbin/modprobe -r openvswitch //删除模块；删除模块时需要加‘-r’参数

## 内核模块Openvswitch.ko的加载

OVS是内核态和用户态配合工作的，所以首先要加载内核态模块Openvswitch.ko。

在datapath/datapath.c中会调用module\_init(dp\_init);来初始化内核模块。

其中比较重要的是调用了dp\_register\_genl()，这个就是注册netlink函数，从而用户态进程ovs-vswitchd可以通过netlink调用内核。

这里dp\_genl\_families由四个netlink的family组成

static struct genl\_family \*dp\_genl\_families[] = {

&dp\_datapath\_genl\_family,

&dp\_vport\_genl\_family,

&dp\_flow\_genl\_family,

&dp\_packet\_genl\_family,

};

可以看出，在内核中，包含对datapath的操作，例如OVS\_DP\_CMD\_NEW，对虚拟端口vport的操作，例如OVS\_VPORT\_CMD\_NEW，对flow流表的操作，例如OVS\_FLOW\_CMD\_NEW，对packet包的操作，例如OVS\_PACKET\_CMD\_EXECUTE。

pr\_info的日志在文件/var/log/kern.log

tail -f kern.log 查看日志

## Linux内核收包流程

参考Linux内核收包流程，函数调用链是：

硬件中断 -->do\_IRQ–>handle\_irq–>e1000\_intr\_msix\_rx–>\_\_napi\_schedule(&adapter->napi)–>

\_\_\_\_napi\_schedule–>\_\_raise\_softirq\_irqoff(NET\_RX\_SOFTIRQ)

do\_IRQ–>irq\_exit–>do\_softirq–>call\_softirq–>\_\_do\_softirq–>

net\_rx\_action->e1000e\_poll–>e1000\_receive\_skb->napi\_gro\_receive–>

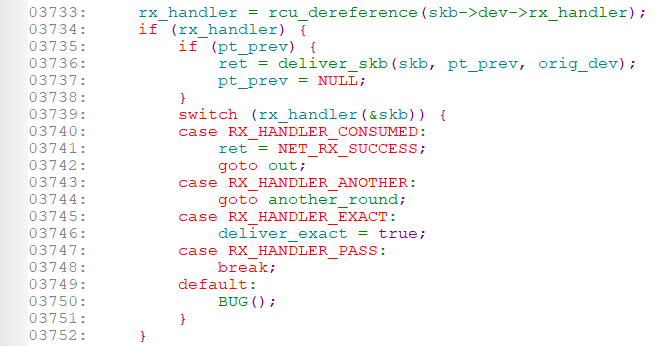
netif\_receive\_skb–>\_\_netif\_receive\_skb–>\_\_netif\_receive\_skb\_core–>

deliver\_skb–>ip\_rcv–>NF\_HOOK(NF\_INET\_PRE\_ROUTING)–>

ip\_rcv\_finish–>dst\_input–>ip\_local\_deliver–>

NF\_HOOK(NF\_INET\_LOCAL\_IN)–>ip\_local\_deliver\_finish–>ipprot->handler()

如果ovs 或者普通的网桥如何报文在哪里接收呢？通过梳理代码发现上述调用流程中的\_\_netif\_receive\_skb\_core函数有个rx\_handler，ovs bridge等都是通过设置该函数指针实现对报文的接管。



Linux-4.1.30(net/core/dev.c \_\_netif\_receive\_skb\_core)

梳理流程发现ovs的rx\_handler指针是netdev\_frame\_hook；bridge的为br\_handle\_frame

虚拟机使用ovs场景下，流量从物理网卡转发到tap设备过程：

\_\_netif\_receive\_skb

    \_\_netif\_receive\_skb\_core

        rx\_handler()   (ovs场景下，通过netdev\_rx\_handler\_register将rx\_handler注册为netdev\_frame\_hook)

            netdev\_frame\_hook

                netdev\_port\_receive

                    ovs\_vport\_receive

                        ovs\_dp\_process\_packet(查找flow规则是否匹配，不匹配upcall到用户态)

                            ovs\_execute\_actions

                                do\_output

                                    ovs\_vport\_send

                                        vport->ops->send(ops在netdev\_create->ovs\_vport\_alloc时指定为ovs\_netdev\_vport\_ops)

                                            dev\_queue\_xmit

                                                \_\_dev\_xmit\_skb（qdisc入队列）

                                                    net\_tx\_action（触发软中断）

                                                        qdisc\_run

                                                            qdisc\_restart

                                                                sch\_direct\_xmit

                                                                    dev\_hard\_start\_xmit

                                                                        xmit\_one

                                                                            netdev\_start\_xmit

                                                                                \_\_netdev\_start\_xmit

                                                                                    ndo\_start\_xmit（执行tap设备对应的函数）

                                                                                        tap\_netdev\_ops

                                                                                            wake\_up\_interruptible\_poll（唤醒vhost线程）

                                                                                                handle\_rx

## 用户态进程ovs-vswitchd的启动

ovs-vswitchd.c的main函数最终会进入一个while循环，在这个无限循环中，里面最重要的两个函数是bridge\_run()和netdev\_run()。

Openvswitch主要管理两种类型的设备，一个是创建的虚拟网桥，一个是连接到虚拟网桥上的设备。

其中bridge\_run就是初始化数据库中已经创建的虚拟网桥。

bridge\_run会调用static void bridge\_reconfigure(const struct ovsrec\_open\_vswitch \*ovs\_cfg)，其中ovs\_cfg是从ovsdb-server里面读取出来的配置。

在这个函数里面，对于每一个网桥，将网卡添加进去。

HMAP\_FOR\_EACH (br, node, &all\_bridges) {

bridge\_add\_ports(br, &br->wanted\_ports);

shash\_destroy(&br->wanted\_ports);

}

最终会调用dpif\_netlink\_port\_add\_\_在这个函数里面，会调用netlink的API，命令为OVS\_VPORT\_CMD\_NEW。

## 内核模块监听网卡

ovs-vswitchd启动的时候，将虚拟网卡添加到虚拟交换机上的时候，会调用netlink的OVS\_VPORT\_CMD\_NEW命令，因而会调用函数ovs\_vport\_cmd\_new。

它最终会调用ovs\_netdev\_link，其中有下面的代码：

err = netdev\_rx\_handler\_register(vport->dev, netdev\_frame\_hook,vport);

注册一个方法叫做netdev\_frame\_hook，每当网卡收到包的时候，就调用这个方法。

<https://blog.csdn.net/kklvsports/article/details/84136047>

<https://www.cnblogs.com/liuhongru/p/11134435.html>

<https://www.sdnlab.com/18668.html>

ovs-vswitchd启动的时候，将虚拟网卡添加到虚拟交换机上的时候，会调用netlink的OVS\_VPORT\_CMD\_NEW命令，因而会调用函数ovs\_vport\_cmd\_new。

它最终会调用ovs\_netdev\_link，其中有下面的代码：

err = netdev\_rx\_handler\_register(vport->dev, netdev\_frame\_hook,vport);

(gdb) bt

#0 dpif\_netlink\_port\_add\_\_ (dpif=0x9a9520, name=0x9a4f90 "ubuntu\_br", type=OVS\_VPORT\_TYPE\_INTERNAL, options=0x0, port\_nop=0x7fffffffe0b8) at lib/dpif-netlink.c:748

#1 0x00000000005cb71b in dpif\_netlink\_port\_add\_compat (dpif=0x9a9520, netdev=0x9cafd0, port\_nop=0x7fffffffe0b8) at lib/dpif-netlink.c:876

#2 0x00000000005cb8c9 in dpif\_netlink\_port\_add (dpif\_=0x9a9520, netdev=0x9cafd0, port\_nop=0x7fffffffe0b8) at lib/dpif-netlink.c:922

#3 0x00000000004b2db1 in dpif\_port\_add (dpif=0x9a9520, netdev=0x9cafd0, port\_nop=0x7fffffffe120) at lib/dpif.c:584

#4 0x000000000043cca5 in port\_add (ofproto\_=0x9a80c0, netdev=0x9cafd0) at ofproto/ofproto-dpif.c:3721

#5 0x000000000041ffc2 in ofproto\_port\_add (ofproto=0x9a80c0, netdev=0x9cafd0, ofp\_portp=0x7fffffffe260) at ofproto/ofproto.c:2013

#6 0x000000000040b430 in iface\_do\_create (br=0x9a7a70, iface\_cfg=0x9a5d00, ofp\_portp=0x7fffffffe260, netdevp=0x7fffffffe270, errp=0x7fffffffe278)

at vswitchd/bridge.c:1811

#7 0x000000000040b5e3 in iface\_create (br=0x9a7a70, iface\_cfg=0x9a5d00, port\_cfg=0x9a6410) at vswitchd/bridge.c:1849

#8 0x0000000000408ad5 in bridge\_add\_ports\_\_ (br=0x9a7a70, wanted\_ports=0x9a7b50, with\_requested\_port=false) at vswitchd/bridge.c:937

#9 0x0000000000408b67 in bridge\_add\_ports (br=0x9a7a70, wanted\_ports=0x9a7b50) at vswitchd/bridge.c:953

#10 0x0000000000407fa6 in bridge\_reconfigure (ovs\_cfg=0x97b990) at vswitchd/bridge.c:667

#11 0x000000000040f0a7 in bridge\_run () at vswitchd/bridge.c:3044

#12 0x0000000000414b70 in main (argc=6, argv=0x7fffffffe568) at vswitchd/ovs-vswitchd.c:127

## 内核态网络包处理

Openvswitch的内核模块openvswitch.ko会在网卡上注册一个函数netdev\_frame\_hook，每当有网络包到达网卡的时候，这个函数就会被调用。

Openvswitch-2.12.0

static rx\_handler\_result\_t netdev\_frame\_hook(struct sk\_buff \*\*pskb)

{

struct sk\_buff \*skb = \*pskb;

if (unlikely(skb->pkt\_type == PACKET\_LOOPBACK))

return RX\_HANDLER\_PASS;

#ifndef USE\_UPSTREAM\_TUNNEL

netdev\_port\_receive(skb, NULL);

#else

netdev\_port\_receive(skb, skb\_tunnel\_info(skb));

#endif

return RX\_HANDLER\_CONSUMED;

}

在netdev\_port\_receive这个函数里面，首先查找vport。

ovs\_vport\_receive要在内核态匹配流表，首先需要调用ovs\_flow\_key\_extract，从包的正文中提取key的值。

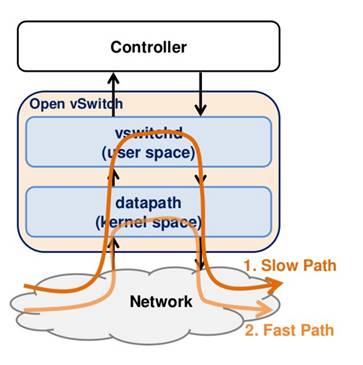
接下来就是要调用ovs\_dp\_process\_packet了。

这个函数首先在内核里面的流表中查找符合key的flow，也即ovs\_flow\_tbl\_lookup\_stats，如果找到了，很好说明用户态的流表已经放入内核，则走fast path就可了。于是直接调用ovs\_execute\_actions，执行这个key对应的action。

如果不能找到，则只好调用ovs\_dp\_upcall，让用户态去查找流表。会调用static int queue\_userspace\_packet(struct datapath \*dp, struct sk\_buff \*skb, const struct sw\_flow\_key \*key, const struct dp\_upcall\_info \*upcall\_info)

它会调用err = genlmsg\_unicast(ovs\_dp\_get\_net(dp), user\_skb, upcall\_info->portid);通过netlink将消息发送给用户态。在用户态，有线程监听消息，一旦有消息，则触发udpif\_upcall\_handler。

Slow Path & Fast Path



Slow Path:

当Datapath找不到flow rule对packet进行处理时

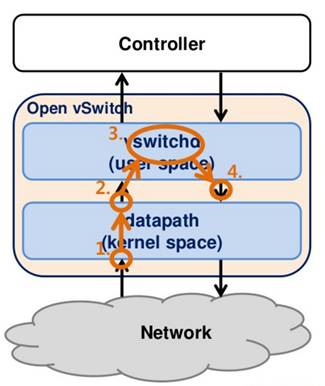
Vswitchd使用flow rule对packet进行处理。

Fast Path:

将slow path的flow rule放在内核态，对packet进行处理

Unknown Packet Processing

Datapath使用flow rule对packet进行处理，如果没有，则有vswitchd使用flow rule进行处理

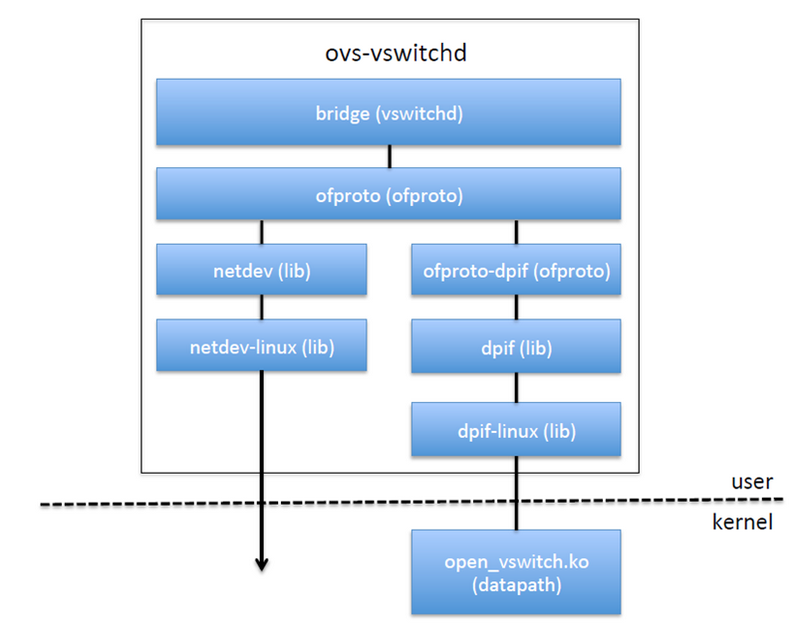


从Device接收Packet交给事先注册的event handler进行处理

接收Packet后识别是否是unknown packet，是则交由upcall处理

vswitchd对unknown packet找到flow rule进行处理

将Flow rule发送给datapath



## 用户态处理包

当内核无法查找到流表项的时候，则会通过upcall来调用用户态ovs-vswtichd中的flow table。

会调用ofproto-dpif-upcall.c中的udpif\_upcall\_handler函数。

(1) 首先读取upcall调用static int upcall\_receive(struct upcall \*upcall, const struct dpif\_backer \*backer, const struct dp\_packet \*packet, enum dpif\_upcall\_type type, const struct nlattr \*userdata, const struct flow \*flow, const unsigned int mru, const ovs\_u128 \*ufid, const unsigned pmd\_id)

(2) 其次提取包头调用void flow\_extract(struct dp\_packet \*packet, struct flow \*flow)，提取出的flow如下：

/\* L2, Order the same as in the Ethernet header! (64-bit aligned) \*/

struct eth\_addr dl\_dst; /\* Ethernet destination address. \*/

struct eth\_addr dl\_src; /\* Ethernet source address. \*/

ovs\_be16 dl\_type; /\* Ethernet frame type. \*/

ovs\_be16 vlan\_tci; /\* If 802.1Q, TCI | VLAN\_CFI; otherwise 0. \*/

ovs\_be32 mpls\_lse[ROUND\_UP(FLOW\_MAX\_MPLS\_LABELS, 2)]; /\* MPLS label stack

(with padding). \*/

/\* L3 (64-bit aligned) \*/

ovs\_be32 nw\_src; /\* IPv4 source address. \*/

ovs\_be32 nw\_dst; /\* IPv4 destination address. \*/

struct in6\_addr ipv6\_src; /\* IPv6 source address. \*/

struct in6\_addr ipv6\_dst; /\* IPv6 destination address. \*/

ovs\_be32 ipv6\_label; /\* IPv6 flow label. \*/

uint8\_t nw\_frag; /\* FLOW\_FRAG\_\* flags. \*/

uint8\_t nw\_tos; /\* IP ToS (including DSCP and ECN). \*/

uint8\_t nw\_ttl; /\* IP TTL/Hop Limit. \*/

uint8\_t nw\_proto; /\* IP protocol or low 8 bits of ARP opcode. \*/

struct in6\_addr nd\_target; /\* IPv6 neighbor discovery (ND) target. \*/

struct eth\_addr arp\_sha; /\* ARP/ND source hardware address. \*/

struct eth\_addr arp\_tha; /\* ARP/ND target hardware address. \*/

ovs\_be16 tcp\_flags; /\* TCP flags. With L3 to avoid matching L4. \*/

ovs\_be16 pad3; /\* Pad to 64 bits. \*/

/\* L4 (64-bit aligned) \*/

ovs\_be16 tp\_src; /\* TCP/UDP/SCTP source port/ICMP type. \*/

ovs\_be16 tp\_dst; /\* TCP/UDP/SCTP destination port/ICMP code. \*/

ovs\_be32 igmp\_group\_ip4; /\* IGMP group IPv4 address.

\* Keep last for BUILD\_ASSERT\_DECL below. \*/

(3) 然后调用static int process\_upcall(struct udpif \*udpif, struct upcall \*upcall, struct ofpbuf \*odp\_actions, struct flow\_wildcards \*wc)来处理upcall。

对于MISS\_UPCALL，调用static void upcall\_xlate(struct udpif \*udpif, struct upcall \*upcall, struct ofpbuf \*odp\_actions, struct flow\_wildcards \*wc)

会调用enum xlate\_error xlate\_actions(struct xlate\_in \*xin, struct xlate\_out \*xout)

在这个函数里面，会在flow table里面查找rule

ctx.rule = rule\_dpif\_lookup\_from\_table( ctx.xbridge->ofproto, ctx.tables\_version, flow, xin->wc, ctx.xin->resubmit\_stats, &ctx.table\_id, flow->in\_port.ofp\_port, true, true);

找到rule之后，调用static void do\_xlate\_actions(const struct ofpact \*ofpacts, size\_t ofpacts\_len, struct xlate\_ctx \*ctx)在这个函数里面，根据action的不同，修改flow的内容。

(4) 最后调用static void handle\_upcalls(struct udpif \*udpif, struct upcall \*upcalls, size\_t n\_upcalls)将flow rule添加到内核中的datapath

他会调用void dpif\_operate(struct dpif \*dpif, struct dpif\_op \*\*ops, size\_t n\_ops)，他会调用dpif->dpif\_class->operate(dpif, ops, chunk);

会调用dpif\_netlink\_operate()

会调用netlink修改内核中datapath的规则。

case DPIF\_OP\_FLOW\_PUT:

put = &op->u.flow\_put;

dpif\_netlink\_init\_flow\_put(dpif, put, &flow);

if (put->stats) {

flow.nlmsg\_flags |= NLM\_F\_ECHO;

aux->txn.reply = &aux->reply;

}

dpif\_netlink\_flow\_to\_ofpbuf(&flow, &aux->request);

break;

## vswitchd和ovsdb通信

* + 1. 建立连接

ovsdb\_idl\_create建立一个ovsdb的连接session，形如：idl =ovsdb\_idl\_create(remote, &ovsrec\_idl\_class, true, true)。

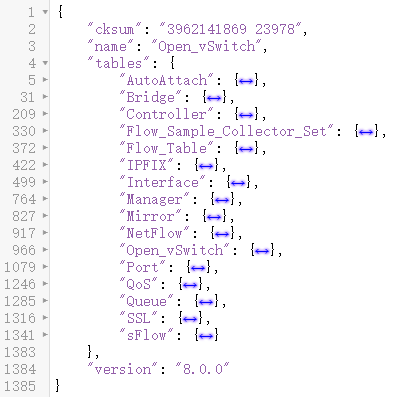
remote类似：unix:/var/run/openvswitch/db.sock。

后续使用该idl（struct ovsdb\_idl）和ovsdb通信，更新数据。

* + 1. 数据库组织形式

数据库名称Open\_vSwitch

其下有许多表



* + 1. 要

sudo ovsdb-client list-dbs

Open\_vSwitch /\* 两个数据库 \*/

\_Server

/\* 每个数据库里的表 \*/

sudo ovsdb-client list-tables \_Server

Table

--------

Database

sudo ovsdb-client list-tables Open\_vSwitch

Table

-------------------------

Controller

Bridge

Queue

IPFIX

NetFlow

Open\_vSwitch

QoS

Port

sFlow

SSL

Flow\_Sample\_Collector\_Set

Mirror

Flow\_Table

Interface

AutoAttach

Manager

/\* 数据库 \_Server 表 Database 里的每一列 \*/

sudo ovsdb-client list-columns \_Server Database

Column Type

--------- -------------------------------------------------------------------

\_version "uuid"

model {"key":{"enum":["set",["clustered","standalone"]],"type":"string"}}

name "string"

connected "boolean"

leader "boolean"

schema {"key":"string","min":0}

\_uuid "uuid"

cid {"key":"uuid","min":0}

sid {"key":"uuid","min":0}

index {"key":"integer","min":0}