抽丝剥茧：生产环境中负载均衡产品DPDK问题的解决

俞文俊 DPDK与SPDK开源社区 1周前

ULB4是UCloud自主研发的基于DPDK的高可用四层负载均衡产品，转发能力接近线速；DPDK则是一个高性能的开源数据面开发套件。ULB4作为用户应用的全局入口，在大流量多元化场景下保证用户业务的持续稳定至关重要，这也是UCloud网络产品团队的技术使命。尤其现网单个ULB集群承载带宽已达10G，包量83万PPS，运行环境复杂，即使面临突发因素(比如触发未知BUG)，我们也要设法保证产品正常工作，避免产生严重影响。

近期，我们在ULB4的线上环境中，发现了一个DPDK的发包异常现象，由于整个ULB产品为集群架构，该异常并未导致用户服务不可用。但为了任何时刻都能保证用户服务的足够稳定，团队通过GDB、报文导出工具、生产环境流量镜像等手段，从现网GB级流量中捕获异常报文，再结合DPDK源码分析，定位到原因出自DPDK本身的BUG并修复解决。期间未对用户业务造成影响，进一步保证了UCloud数万ULB实例的稳定运行。

本文将从问题现象着手，抽丝剥茧，详述问题定位、分析与解决全过程，希望能为ULB用户和DPDK开发者提供参考与启迪。

问题背景

在12月初一向稳定的ULB4集群中突然出现了容灾，某台ULB4服务器工作异常被自动移出了集群。当时的现象是：

转发面服务监控到网卡接收方向流量正常，但是发送方向流量为0，重启转发面服务后又可以正常收发，同时集群其他机器也会不定期出现异常情况。对用户业务而言，会出现少量连接轻微抖动，随后迅速恢复。

下面是整个问题的处理过程，我们在此过程中做出种种尝试，最终结合DPDK源码完成分析和解决，后续也准备将自研的报文导出工具开源共享。

问题定位与分析

ULB4集群一直很稳定地工作，突然陆续在集群的不同机器上出现同样的问题，并且机器恢复加入集群后，过了一段时间又再次出现同样的问题。根据我们的运营经验，初步猜测是某种异常报文触发了程序BUG。但是，面对GB级流量如何捕获到异常报文？又如何在不影响业务情况下找出问题呢？

1、GDB调试报文，发现疑点

想要知道整个程序为什么不发包，最好的办法就是能够进入到程序中去看看具体的执行过程。对于DPDK用户态程序来说，GDB显然是一个好用的工具。我们在发包程序逻辑中设置断点，并通过disassemble命令查看该函数的执行逻辑，反汇编之后足足有七百多行。（该函数中调用的很多函数都使用了inline修饰，导致该函数在汇编之后指令特别多）

```
(gdb) disass
Dump of assembler code for function i40e_xmit_pkts:
=> 0x0000000000ea7750 <+0>:     push   %r15
   0x0000000000ea7752 <+2>:     mov    %rdi,%rax
   0x0000000000ea7755 <+5>:     push   %r14
   0x0000000000ea7757 <+7>:     push   %r13
   0x0000000000ea7759 <+9>:     push   %r12
```

```
   0x0000000000ea8450 <+3328>:  je     0xea78f8 <i40e_xmit_pkts+424>
   0x0000000000ea8456 <+3334>:  jmpq   0xea7970 <i40e_xmit_pkts+544>
   0x0000000000ea845b <+3339>:  lock subw $0x1,0x14(%r9)
   0x0000000000ea8462 <+3346>:  jne    0xea7902 <i40e_xmit_pkts+434>
   0x0000000000ea8468 <+3352>:  nopl   0x0(%rax,%rax,1)
   0x0000000000ea8470 <+3360>:  jmpq   0xea81f5 <i40e_xmit_pkts+2725>
End of assembler dump.
(gdb) ni
0x0000000000ea7752 in i40e_xmit_pkts ()
(gdb)
```

结合对应DPDK版本的源码，单条指令一步步执行。在多次尝试之后，发现每次都会在下图所示的地方直接返回。

```
static inline int
i40e_xmit_cleanup(struct i40e_tx_queue *txq)
{
    struct i40e_tx_entry *sw_ring = txq->sw_ring;
    volatile struct i40e_tx_desc *txd = txq->tx_ring;
    uint16_t last_desc_cleaned = txq->last_desc_cleaned;
    uint16_t nb_tx_desc = txq->nb_tx_desc;
    uint16_t desc_to_clean_to;
    uint16_t nb_tx_to_clean;

    desc_to_clean_to = (uint16_t)(last_desc_cleaned + txq->tx_rs_thresh);
    if (desc_to_clean_to >= nb_tx_desc)
        desc_to_clean_to = (uint16_t)(desc_to_clean_to - nb_tx_desc);

    desc_to_clean_to = sw_ring[desc_to_clean_to].last_id;
    if ((txd[desc_to_clean_to].cmd_type_offset_bsz &
            rte_cpu_to_le_64(I40E_TXD_QW1_DTYPE_MASK)) !=
            rte_cpu_to_le_64(I40E_TX_DESC_DTYPE_DESC_DONE)) {
        PMD_TX_FREE_LOG(DEBUG, "TX descriptor %4u is not done "
            "(port=%d queue=%d)", desc_to_clean_to,
                txq->port_id, txq->queue_id);
        return -1;
    }
```

大致流程是i40e_xmit_pkts()在发送的时候，发现发送队列满了就会去调用i40e_xmit_cleanup()清理队列。DPDK中网卡在发送完数据包后会去回写特定字段，表明该报文已经发送，而驱动程序去查看该字段就可以知道这个报文是否已经被发过。此处的问题就是驱动程序认为该队列中的报文始终未被网卡发送出去，后续来的报文将无法加入到队列而被直接丢弃。

至此，直接原因已经找到，**就是网卡因为某种原因不发包或者没能正确回写特定字段，导致驱动程序认为发送队列始终处于队列满的状态，而无法将后续的报文加入发送队列。**

那么为什么出现队列满？异常包是否相关呢？带着这个疑问，我们做了第二个尝试。

2、一键还原网卡报文

队列满，而且后面的报文一直加不进去，说明此时队列里面的报文一直卡在那。既然我们猜测可能是存在异常报文，那么有没有可能异常报文还在队列里面呢？如果可以把当前队列里面的报文全部导出来，那就可以进一步验证我们的猜测了。

基于对DPDK的深入研究，我们根据以下步骤导出报文。

- 我们看i40e_xmit_pkts()函数，会发现第一个参数就是发送队列，所以我们可以获取到队列的信息。

```
uint16_t
i40e_xmit_pkts(void *tx_queue, struct rte_mbuf **tx_pkts, uint16_t nb_pkts)
{
    struct i40e_tx_queue *txq;
    struct i40e_tx_entry *sw_ring;
    struct i40e_tx_entry *txe, *txn;
    volatile struct i40e_tx_desc *txd;
```

- 如下图所示，在刚进入断点的时候，查看寄存器信息，以此来获得该函数对应的参数。

```
# gdb attach 20896

(gdb) b i40e_xmit_pkts
Breakpoint 1 at 0xeacc80
(gdb) c
Continuing.

Breakpoint 1, 0x0000000000eacc80 in i40e_xmit_pkts ()
(gdb) info reg
rax            0x7f993094a300   140295921771264
rbx            0x0      0
rcx            0x0      0
rdx            0x1      1
rsi            0x7f993094eb90   140295921789840
rdi            0x7f993094a140   140295921770816
rbp            0x17f1b40        0x17f1b40 <rte_eth_devices>
rsp            0x7f9d34154a38   0x7f9d34154a38
r8             0x200    512
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x7f993094eb90   140295921789840
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0xeacc80 0xeacc80 <i40e_xmit_pkts>
eflags         0x3246   [ PF ZF IF #12 #13 ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

- 当我们打印该队列的消息时，却发现没有符号信息，此时我们可以如下图所示去加载编译时候生成的 i40e_rxtx.o 来获取对应符号信息。

```
(gdb) set print pretty on
(gdb) print *((struct i40e_tx_queue*)0x7f993094a140)
No struct type named i40e_tx_queue.
(gdb) dele
Delete all breakpoints? (y or n) y

(gdb) add-symbol-file /root/tmp/i40e_rxtx.o 0
add symbol table from file "/root/tmp/i40e_rxtx.o" at
        .text_addr = 0x0
(y or n) y
Reading symbols from /root/tmp/i40e_rxtx.o...done.

(gdb) print *((struct i40e_tx_queue*)0x7f993094a140)
$1 = {
  nb_tx_desc = 4096,
  tx_ring_phys_addr = 159833728,
```

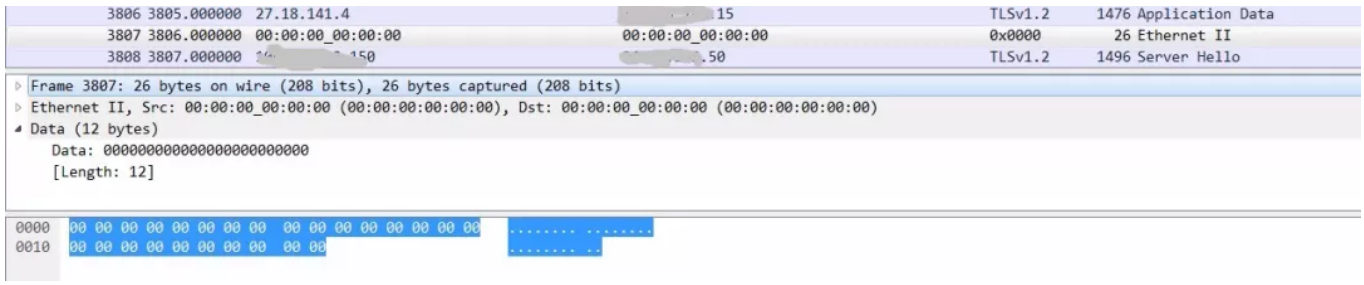
- 在得到队列信息后，我们使用GDB的dump命令将整个队列中所有的报文全部按队列中的顺序导出，对每个报文按序号命名。

```
(gdb) print *((struct i40e_tx_queue*)0x7f993094a140)
$1 = {
  nb_tx_desc = 4096,        发送队列描述符容量
  tx_ring_phys_addr = 159833728,
  tx_ring = 0x7f9d3186de80,
  sw_ring = 0x7f9d3185de40,
  tx_tail = 3778,
  qtx_tail = 0x7f992f907004 "\302\016",
  nb_tx_used = 31,
  last_desc_cleaned = 3778,
  nb_tx_free = 0,
  tx_free_thresh = 32,
  tx_rs_thresh = 32,
  pthresh = 32 ' ',
  hthresh = 0 '\000',
  wthresh = 0 '\000',
  port_id = 0 '\000',
  queue_id = 0,
  reg_idx = 1,
  txq_flags = 0,
  vsi = 0x7f993094ab40,
  tx_next_dd = 31,
  tx_next_rs = 31,
  q_set = 1,
  tx_deferred_start = 0,
  dcb_tc = 0 '\000'
}

(gdb) set $idx = 0
(gdb) set $mbuf_addr = ((struct rte_mbuf *)((struct i40e_tx_queue*)0x7f993094a140)->sw_ring[$idx].mbuf)->buf_addr
(gdb) set $mbuf_data_off = ((struct rte_mbuf *)((struct i40e_tx_queue*)0x7f993094a140)->sw_ring[$idx].mbuf)->data_off
(gdb) set $mbuf_pkt_len = ((struct rte_mbuf *)((struct i40e_tx_queue*)0x7f993094a140)->sw_ring[$idx].mbuf)->pkt_len

(gdb) dump binary memory /root/tmp/bins/0.bin $mbuf_addr+$mbuf_data_off $mbuf_addr+$mbuf_data_off+$mbuf_pkt_len
```

- 此时导出的报文还是原始的报文，我们无法使用wireshark方便地查看报文信息。为此如下图所示，我们使用libpcap库写了个简单的小工具转换成wireshark可以解析的pcap文件。

```
    hdr.ts.tv_sec = 0;
    hdr.ts.tv_usec = 0;
    hdr.caplen = len;
    hdr.len = len;

    handler = pcap_open_dead(1, 65535);
    pdumper = pcap_dump_open(handler, pcap_name);

    for (i = 0; i < bin_nums; i++) {

        memset(tmp, 0, sizeof(tmp));
        sprintf(tmp, "%s/%d.bin", bin_dir, i);

        memset(buf, 0, sizeof(buf));
        hdr.ts.tv_sec = i;

        len = getbin(buf, sizeof(buf), tmp);
        hdr.caplen = len;
        hdr.len = len;

        pcap_dump((u_char*)pdumper, &hdr, buf);
    }

    pcap_dump_flush(pdumper);

    if (handler) {
        free(handler);
    }

    pcap_dump_close(pdumper);

    return;
}
```

果然，如下图所示，在导出的所有报文中包含了一个长度为26字节，但内容为全0的报文。这个报文看上去十分异常，似乎初步验证了我们的猜测：



为了提高在排查问题时导出报文的速度，我们写了一个报文一键导出工具，可以在异常时一键导出所有的报文并转成pcap格式。

在多次导出报文后，我们发现一个规律：每次都会有一个长度为26字节但是全0的报文，而且在其前面都会有一个同样长度的报文，且每次源IP地址网段都来自于同一个地区。

三、流量镜像，确认异常包

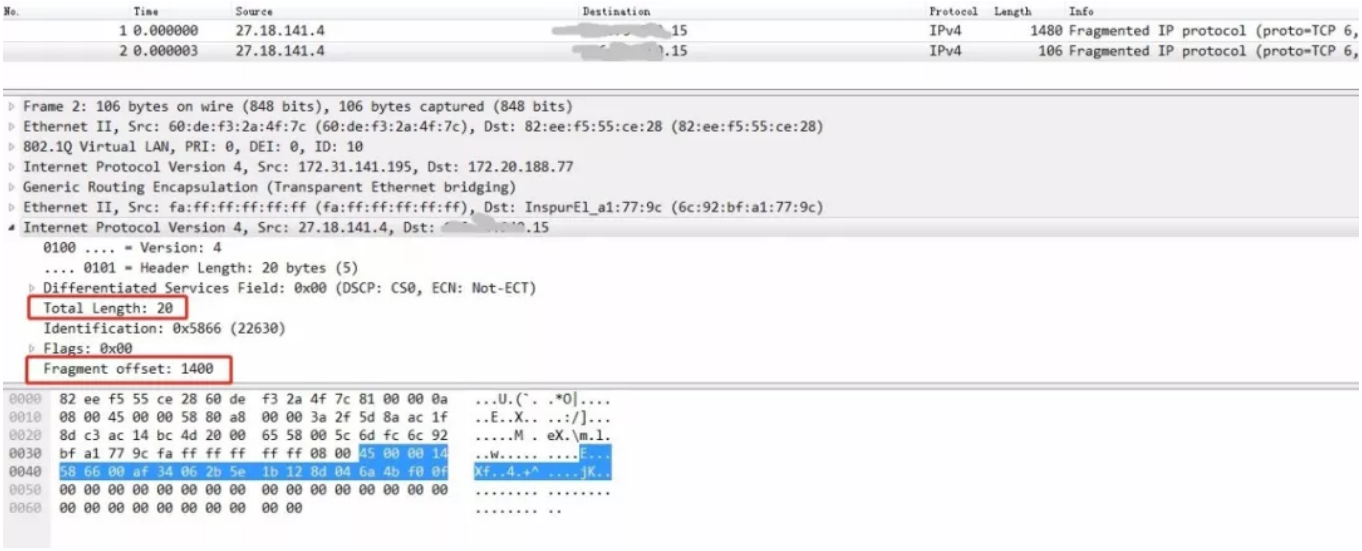
第二步结论让整个排查前进了一大步，但是队列包是经过一系列程序处理的，并不是真正的原始业务报文。不达目的不罢休，关键时刻还是要上镜像抓包，于是当晚紧急联系网络运维同事在交换机上配置port-mirroring（端口镜像），将发往ULB4集群的流量镜像到一个空闲服务器上进行镜像抓包。当然，镜像服务器还需要做特殊配置，如下：

1. 设置网卡混杂模式，用于收取镜像流量（ifconfig net2 promisc）。
2. 关闭GRO功能（ethtool -K net2 gro off），用于收取最原始的报文，防止Linux的GRO功能提前将报文进行组装。

根据异常IP的地域特性，我们针对性抓取了部分源IP段的流量。

参考命令：nohup tcpdump -i net2 -s0 -w %Y%m%d_%H-%M-%S.pcap -G 1800 "proto gre and (((ip[54:4]&0x11223000)==0x11223000) or ((ip[58:4]&0x11223000)==0x11223000))" &

经过多次尝试后，功夫不负有心人，故障出现了，经过层层剥离筛选，找到了如下报文：



这是IP分片报文，但是奇怪的是IP分片的第二片只有IP头。经过仔细比对，这两个报文合在一起就是导出队列报文中的那两个连在一起的报文。后26字节和全0报文完全吻合。

我们知道在TCP/IP协议中，如果发送时一个IP报文长度超过了MTU，将会触发IP分片，会被拆成多个小的分片报文进行发送。正常情况下，所有的分片肯定都是携带有数据的。但是这一个分片报文就很异常，报文的总长度是20，也就是说只有一个IP头，后面不再携带任何信息，这样的报文是没有任何意义的。这个报文还因为长度太短在经过交换机后被填充了26字节的0。

至此，我们最终找到了这个异常报文，也基本验证了我们的猜测。但是还需要去实际验证是否为这种异常报文导致。（从整个报文的交互来看，这一片报文本来是设置了不可分片的TCP报文，但是在经过某个公网网关后被强制设定了允许分片，并且分片出了这种异常的形式。）

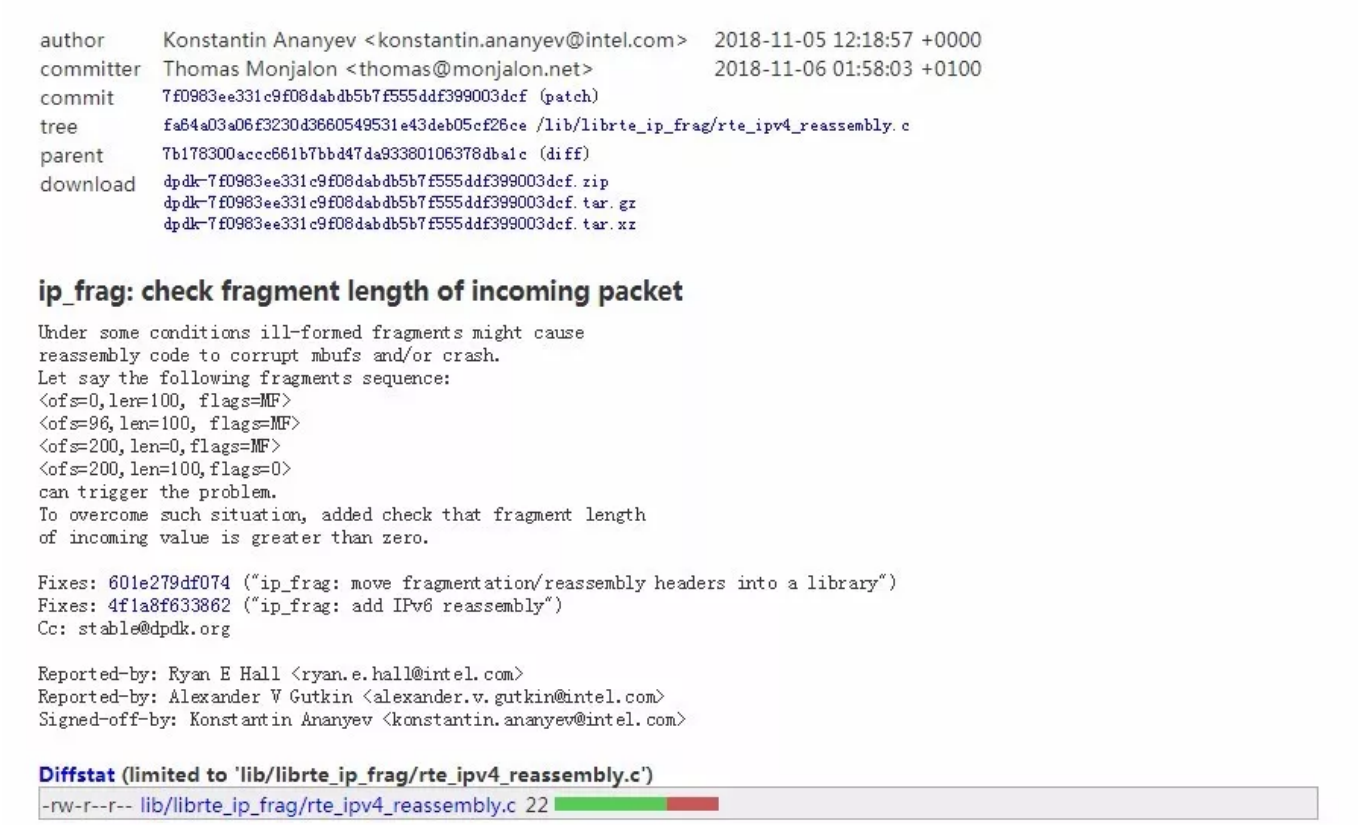
四、解决方案

如果确实是这个异常报文导致的，那么只要在收包时对这种异常报文进行检查然后丢弃就可以了。于是，我们修改DPDK程序，丢弃这类报文。作为验证，先发布了一台线上服务器，经过1天运行再也没有出现异常容灾情况。既然问题根因已经找到，正是这种异常报文导致了DPDK工作异常，后续就可以按灰度全网发布了。

五、DPDK社区反馈

本着对开源社区负责任的态度，我们准备将BUG向DPDK社区同步。对比最新的commit后，找到11月6日提交的一个commit，情况如出一辙，如下：

ip_frag: check fragment length of incoming packet



author Konstantin Ananyev <konstantin.ananyev@intel.com> 2018-11-05 12:18:57 +0000
committer Thomas Monjalon <thomas@monjalon.net> 2018-11-06 01:58:03 +0100
commit 7f0983ee331c9f08dabdb5b7f555ddf399003dcf (patch)
tree fa64a03a06f3230d3660549531e43deb05cf26ce /lib/librte_ip_frag/rte_ipv4_reassembly.c
parent 7b178300accc661b7bbd47da93380106378dba1c (diff)
download dpdk-7f0983ee331c9f08dabdb5b7f555ddf399003dcf.zip
dpdk-7f0983ee331c9f08dabdb5b7f555ddf399003dcf.tar.gz
dpdk-7f0983ee331c9f08dabdb5b7f555ddf399003dcf.tar.xz

ip_frag: check fragment length of incoming packet

Under some conditions ill-formed fragments might cause
reassembly code to corrupt mbufs and/or crash.
Let say the following fragments sequence:
<ofs=0,len=100, flags=MF>
<ofs=96,len=100, flags=MF>
<ofs=200,len=0,flags=MF>
<ofs=200,len=100,flags=0>
can trigger the problem.
To overcome such situation, added check that fragment length
of incoming value is greater than zero.

Fixes: 601e279df074 ("ip_frag: move fragmentation/reassembly headers into a library")
Fixes: 4f1a8f633862 ("ip_frag: add IPv6 reassembly")
Cc: stable@dpdk.org

Reported-by: Ryan E Hall <ryan.e.hall@intel.com>
Reported-by: Alexander V Gutkin <alexander.v.gutkin@intel.com>
Signed-off-by: Konstantin Ananyev <konstantin.ananyev@intel.com>

Diffstat (limited to 'lib/librte_ip_frag/rte_ipv4_reassembly.c')

-rw-r--r-- lib/librte_ip_frag/rte_ipv4_reassembly.c 22

DPDK 18.11最新发布的版本中，已对此进行了修复，和我们处理逻辑一致，也是丢弃该异常报文。

```
@@ -129,6 +131,12 @@ rte_ipv4_frag_reassemble_packet(struct rte_ip_frag_tbl *tbl,
                tbl, tbl->max_cycles, tbl->entry_mask, tbl->max_entries,
                tbl->use_entries);

+       /* check that fragment length is greater then zero. */
+       if (ip_len <= 0) {
+               IP_FRAG_MBUF2DR(dr, mb);
+               return NULL;
+       }
+
        /* try to find/add entry into the fragment's table. */
        if ((fp = ip_frag_find(tbl, dr, &key, tms)) == NULL) {
                IP_FRAG_MBUF2DR(dr, mb);
```

复盘和总结

处理完所有问题后，我们开始做整体复盘。

一、ULB无法发包的成因总结

ULB4无法发包的整个产生过程如下：

1. DPDK收到分片报文中的第一片，将其缓存下来等待后续分片；
2. 第二片只有IP头的异常分片到来，DPDK按照正常的报文处理逻辑进行处理，并没有进行检查丢弃，于是两片报文的rte_mbuf结构被链在一起，组成了一个链式报文返回给ULB4；
3. 这样的报文被ULB4接收后，因为整个报文的总长度并没有达到需要分片的长度，所以ULB4直接调用DPDK的发送接口发送出去；
4. DPDK没有对这种异常报文进行检查，而是直接调用相应的用户态网卡驱动直接将报文发送出去；
5. 用户态网卡驱动在发送这样的异常报文时触发了网卡tx hang；
6. 触发tx hang后，网卡不再工作，驱动队列中报文对应的发送描述符不再被网卡正确设置发送完成标记；
7. 后续的报文持续到来，开始在发送队列中积压，最终将整个队列占满，再有报文到来时将被直接丢弃。

二、为什么异常报文会触发网卡tx hang

首先我们看下DPDK中跟网卡发送报文相关的代码。

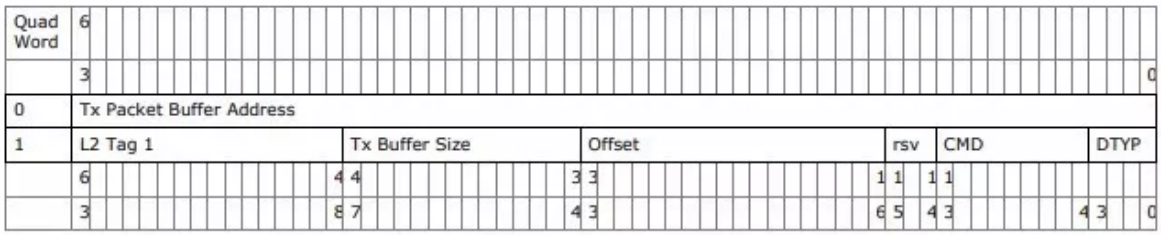
```
uint16_t
i40e_xmit_pkts(void *tx_queue, struct rte_mbuf **tx_pkts, uint16_t nb_pkts) {
    struct i40e_tx_entry *txe, *txn;
    volatile struct i40e_tx_desc *txd;
    volatile struct i40e_tx_desc *txr = txq->tx_ring;
    ...
    m_seg = tx_pkt;      // 程序传递过来的rte_mbuf此时开始设置各种对应关系
    do {
        txd = &txr[tx_id];
        txn = &sw_ring[txe->next_id];

        if (txe->mbuf)
            rte_pktmbuf_free_seg(txe->mbuf);
        txe->mbuf = m_seg;  // 将rte_mbuf封装到struct i40e_tx_entry结构中

        /* Setup TX Descriptor */
        slen = m_seg->data_len;  // 取出rte_mbuf中对应的数据长度

        /* mb->buf_physaddr + mb->data_off 取出其中数据对应的起始地址 */
        buf_dma_addr = rte_mbuf_data_dma_addr(m_seg);

        /* 以下两步设置网卡发送描述符的起始地址和数据长度。
         * 过程中会去根据网卡的Datasheet设置相应的标记位。
         */
        txd->buffer_addr = rte_cpu_to_le_64(buf_dma_addr);
        txd->cmd_type_offset_bsz = i40e_build_ctob(td_cmd, td_offset, slen, td_tag);

        txe->last_id = tx_last;
        tx_id = txe->next_id;
        txe = txn;
        m_seg = m_seg->next; // 如果该rte_mbuf是链式，就取出后面的进行设置
    } while (m_seg != NULL);
    ...
}
```

```
static inline int
i40e_xmit_cleanup(struct i40e_tx_queue *txq)
{
    struct i40e_tx_entry *sw_ring = txq->sw_ring;
    volatile struct i40e_tx_desc *txd = txq->tx_ring;
    uint16_t last_desc_cleaned = txq->last_desc_cleaned;
    uint16_t nb_tx_desc = txq->nb_tx_desc;
    uint16_t desc_to_clean_to;
    uint16_t nb_tx_to_clean;

    desc_to_clean_to = (uint16_t)(last_desc_cleaned + txq->tx_rs_thresh);
    if (desc_to_clean_to >= nb_tx_desc)
        desc_to_clean_to = (uint16_t)(desc_to_clean_to - nb_tx_desc);

    /*
     * 网卡在发送完成后会去write back发送描述符的特定字段，
     * 此处是i40e_tx_desc.cmd_type_offset_bsz。
     * 驱动会去根据这个字段判断网卡是否已经将报文发出去。
     */
    desc_to_clean_to = sw_ring[desc_to_clean_to].last_id;
    if ((txd[desc_to_clean_to].cmd_type_offset_bsz &
            rte_cpu_to_le_64(I40E_TXD_QW1_DTYPE_MASK)) !=
            rte_cpu_to_le_64(I40E_TX_DESC_DTYPE_DESC_DONE)) {
        PMD_TX_FREE_LOG(DEBUG, "TX descriptor %4u is not done "
            "(port=%d queue=%d)", desc_to_clean_to,
                txq->port_id, txq->queue_id);
        return -1;
    }
}
```

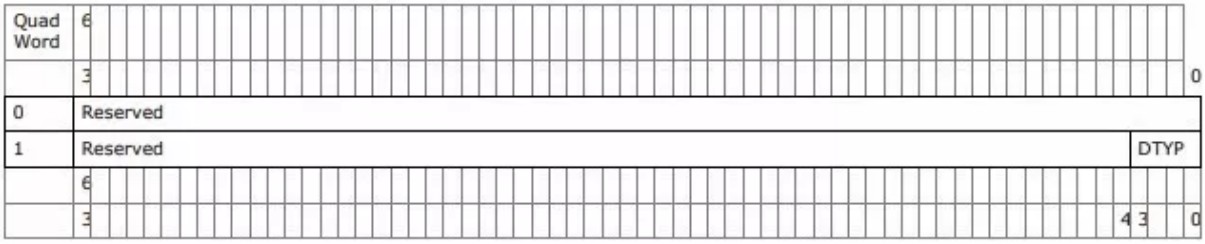
从以上的图中我们可以看到，根据网卡的Datasheet对相关字段进行正确设置非常重要，如果某种原因设置错误，将可能会导致不可预知的后果（具体还是要参考网卡的Datasheet）。

如下图所示，通常网卡对应的Datasheet中会对相应字段进行相关描述，网卡驱动中一般都会有相应的数据结构与其对应。

8.4.2.1.1 Transmit Data Descriptor



8.4.2.1.3 Transmit Descriptors Write Back Format



在有了基本了解后，我们猜想如果直接在程序中手动构造这种类似的异常报文，是否也会导致网卡异常不发包？

答案是肯定的。

如下图所示，我们使用这样的代码片段构成异常报文，然后调用DPDK接口直接发送，很快网卡就会tx hang。

```
struct rte_mbuf * new_pkt = rte_pktmbuf_alloc(packet_pool_);
struct rte_mbuf * new_pkt2 = rte_pktmbuf_alloc(packet_pool_);

new_pkt2->vlan_tci = 0;
new_pkt2->pkt_len = 26;
new_pkt2->data_len = 0;
new_pkt2->data_off = 204;

new_pkt->vlan_tci = 0;
new_pkt->pkt_len = 1200;
new_pkt->data_len = 1200;
new_pkt->data_off = 128;
new_pkt->next = new_pkt2;
```

三、对直接操作硬件的思考

直接操作硬件是一件需要非常谨慎的事情，在传统的Linux系统中，驱动程序一般处于内核态由内核去管理，而且驱动程序代码中可能进行了各种异常处理，因此很少会发生用户程序操作导致硬件不工作的情况。而DPDK因为其自身使用用户态驱动的特点，使得可以在用户态直接操作硬件，同时为了提升性能可能进行了非常多的优化，如果用户自身程序处理出问题就有可能会导致网卡tx hang这样的异常情况发生。

四、工具的价值

我们编写了一键导出DPDK驱动队列报文的工具，这样就可以在每次出现问题时，快速导出网卡驱动发送队列中的所有报文，大大提高了排查效率。这个工具再优化下后，准备在UCloud GitHub上开源，希望对DPDK开发者有所帮助。

写在最后

DPDK作为开源套件，通常情况下稳定性和可靠性不存在什么问题，但是实际的应用场景千变万化，一些特殊情况可能导致DPDK工作异常。虽然发生概率很小，但是DPDK通常在关键的网关位置，一旦出现了问题，哪怕是很少见的问题也将会产生严重影响。

因此技术团队理解其工作原理并对其源码进行分析，同时能够结合具体现象一步步定位出DPDK存在的问题，对提高整个DPDK程序的服务可靠性具有重要意义。值得一提的是，ULB4的高可用集群架构在本次问题的处理过程中发挥了重要作用，在一台不可用的时候，集群中其他机器也可以继续为用户提供可靠服务，有效提升了用户业务的可靠性。

— END —

文章转载自公众号

UCloud技术