# 网络加速

## NAPI

NAPI全称是New API，这是Linux系统针对网络接收的优化。硬件I/O与CPU的交互一般有中断和轮询两种方式。中断的CPU代价较大，但是实时性好，且不需要CPU一直值守，而轮询需要CPU定期查询I/O，需要CPU一直值守，并且不是真正的实时。对于网卡来说，一个繁忙的网络，每次网络数据包到达，如果都采用中断，这样频繁的中断会影响系统的整体效率。而对于一个流量小的网络，如果采用轮询，一个是实时性差，会导致延时（Latency）上升，另一方面CPU需要一直值守，CPU效率低。

NAPI根据不同的场景，采用不同的方式作为CPU和网卡的交互方法，在大网络流量的时候，采用轮询的方式，读取网卡数据，小网络流量的时候则采用中断的方式，从而提高CPU的效率。

## Checksum offload

很多网络协议，例如IP、TCP、UDP都有自己的校验和（checksum）。传统上，校验和的计算（发送数据包）和验证（接收数据包）是通过CPU完成的。这对CPU的影响很大，因为校验和需要每个字节的数据都参与计算。对于一个100G带宽的网络，需要CPU最多每秒计算大约12G的数据。

为了减轻这部分的影响，现在的网卡，都支持校验和的计算和验证。系统内核在封装网络数据包的时候，可以跳过校验和。网卡收到网络数据包之后，根据网络协议的规则，进行计算，再将校验和填入相应的位置。

因为Checksum offload的存在，在用tcpdump之类的抓包分析工具时，有时会发现抓到的包提示校验和错误（checksum incorrect）。tcpdump抓到的网络包就是系统内核发给网卡的网络包，如果校验和放到网卡去计算，那么tcpdump抓到包的时刻，校验和还没有被计算出来，自然看到的是错误的值。

## Scatter/Gather

这项加速只能用于网络数据的发送。Scatter/Gather本身也是操作系统里面一个通用的技术，也叫做vector addressing。简单来说，就是数据在传输的过程中，数据的读取方，不需要从一段连续的内存读取数据，而是可以从多个离散的内存地址读取数据。例如，系统内核在收到应用程序传来的原始数据时，可以保持这段数据不动。之后在另一块内存中计算出各层协议的Header。最后通知网卡驱动，从这两块内存中将数据拷贝过去。SG可以减少不必要的内存拷贝操作。

SG需要Checksum offload的支持，因为现在数据是离散的，系统内核不太容易计算Checksum。

## TSO

TSO全称是TCP Segmentation Offload，它只能用于网络数据的发送。从名字可以看出，这是一个与TCP协议紧密相关的方法。

应用程序可以传递任意长度数据给TCP。TCP位于传输层并不会直接将整段用户数据交给下层协议去传输。因为TCP本身是一个可靠的传输协议，而下层协议，IP/Ethernet都不是可靠的，下层协议在数据传输过程中可能会丢失数据。TCP不仅需要确保传输的可靠性，为了保证效率，还需要尽量提高传输的成功率。TCP的办法就是化整为零，各个击破。

在开始后面的描述之前，先说两个相近且容易混淆的词。一个是Segmentation（分段），一个是Fragmentation（分片）。TCP协议在将用户数据传给IP层之前，会先将大段的数据根据MSS（Maximum Segment Size）分成多个小段，这个过程是Segmentation，分出来的数据是Segments。IP协议因为MTU（Maximum Transmission Unit）的限制，会将上层传过来的并且超过MTU的数据，分成多个分片，这个过程是Fragmentation，分出来的数据是Fragments。这两个过程都是大块的数据分成多个小块数据，区别就是一个在TCP（L4），一个在IP（L3）完成。

接着回来，如果TCP直接传输整段数据给下层协议，假设是15000字节的用户数据，网卡的MTU是1500，考虑到Header，IP层会将数据分成11个IP Fragments在网络上传输，为了描述简单，我们就假设分成了10个IP Fragments。假设每个IP packet的传输成功率是90%，因为TCP协议有自己的校验和，在数据的接收端，IP协议必须将完整的15000字节的用户数据收完，并且拼接传给TCP，才算接收端成功收到数据。这样的话，传输一次成功传输的概率是(90%)^10=34%。一旦TCP接收端没有成功收到数据，发送端就需要重新将整段数据15000字节再发一次。假设发送4次，也就是总共60000字节，传输的成功率能上升到80%。

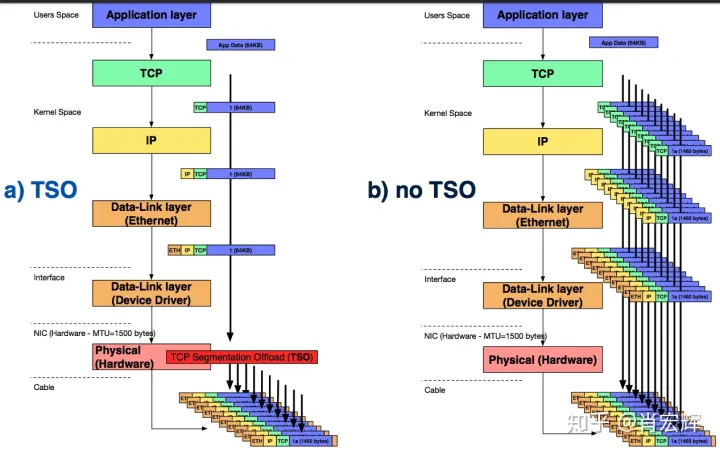
如果TCP协议本身就将数据分成小段，一段一段传输呢？前面说过，TCP是根据MSS完成Segmentation，MSS通常是根据MTU计算，以确保一个TCP Segment不必在IP协议层再进行Fragmentation。为了描述简单，我们还是抛开网络协议的头部，现在TCP将应用层的15000字节数据在自己这里分成了10个Segments。每个Segment对应一个IP packet，成功率还是90%。如果Segment发送失败了，TCP只需要重传当前Segment，之前已经成功发送的TCP Segment不必重传。这样，对于每个Segment，只要发送2次成功率就能达到99%。假设每个Segment发送2次，相应的应用层数据总共发送2次，也就是30000字节，传输的成功率可以达到(99%)^10=90%。也就是说TCP Segmentation之后再传输，需要发送的数据量更少，成功率反而更高。当然实际中，因为TCP Segmentation，会对每个TCP Segment增加TCP 头部，相应传输的数据会更多一点，但是前面的分析结果不受这点数据量的影响。所以，TCP Segmentation对于TCP的可靠来说是必须的。

但同时，它也有自身的缺点。TCP Segmentation之后，相当于对于一段数据，分成了若干个TCP Segments，每个Segment都有自己的TCP头部，这若干个TCP头部，都需要CPU去计算checksum，sequence等。同时，每个TCP Segment还会有自己的IP协议头部，也需要CPU去计算IP协议头部的内容。所以可以预见的是，TCP Segmentation之后，CPU的负担增加了许多。

TSO就是将TCP Segmentation的工作，卸载（offload）到网卡来完成。有了TSO，操作系统只需要传给硬件网卡一个大的TCP数据（当然是包在Ethernet Header和IP Header内，且不超过64K）。网卡会代替TCP/IP协议栈完成TCP Segmentation。这样，就消除了TCP Segmentation带来的CPU负担。

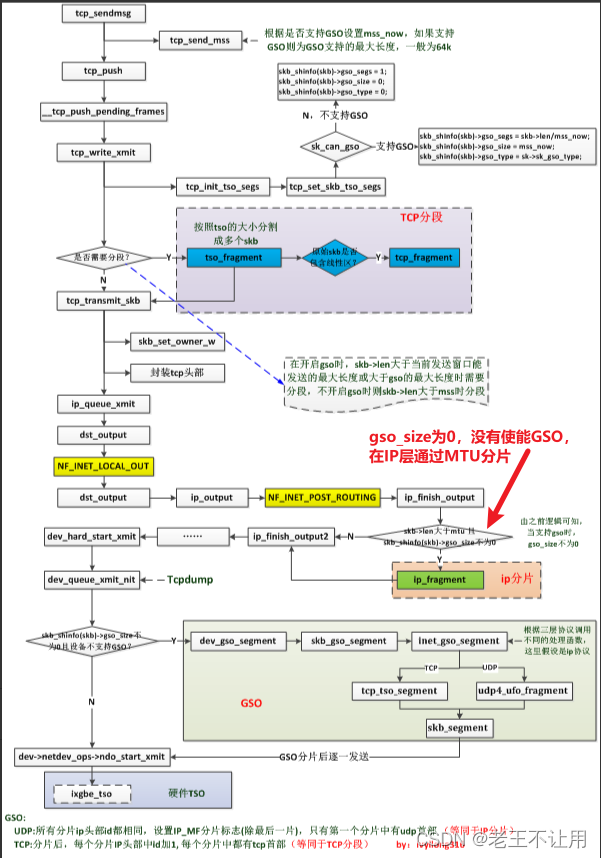
另一个好处在DMA。虽然说每次DMA操作，不需要CPU太多的介入，但是仍然需要CPU配置DMA控制器。DMA的特点在于，无论传输数据的长短，配置工作量是一样的。如果系统内核自己完成TCP Segmentation，那么就有若干个TCP Segments需要通过DMA传给网卡。而采用TSO，因为传输的是一大段数据，只需要配置一次DMA，就可以将数据拷贝到网卡。这也一定程度减轻了CPU的负担。

支持TSO的网卡，仍然会按照TCP/IP协议将网络数据包生成好并发送出去。对于外界系统来说，感受不到TSO的存在。下图是TSO和非TSO下，TCP/IP协议栈对网络数据的处理过程对比。



TSO带来的提升是明显的，一方面，更多的CPU被释放出来完成别的工作，另一方面，网络吞吐量（throughput）不受CPU负荷的影响，如果没有TSO，当CPU性能不好或者CPU本身负荷已经较大时，CPU将来不及处理足够的网络数据，会导致网络吞吐量下降，延时上升。TSO需要SG和Checksum offload的支持。因为TCP/IP协议栈并不知道最终的网络数据包是什么样，自然也没办法完成校验和计算。

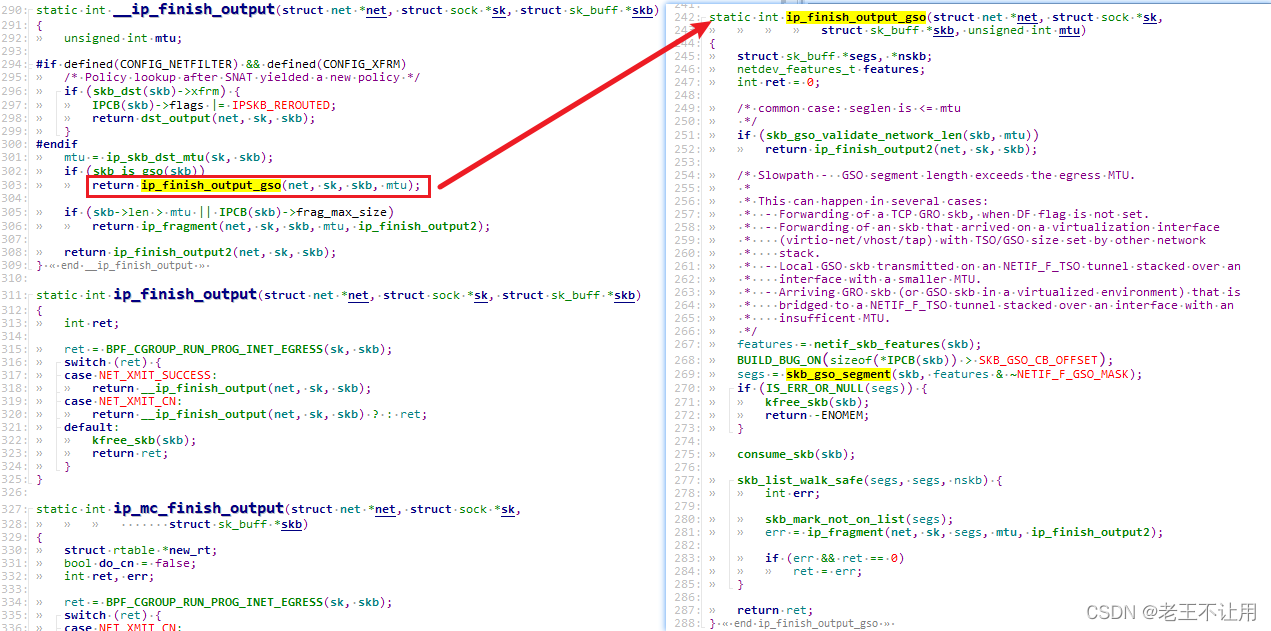
### tso实现





### 5.19







### sk\_can\_gso是否推迟分段

GSO/TSO是否开启是保存在dev->features中，而设备和路由关联，当我们查询到路由后就可以把配置保存在sock中。

比如在tcp\_v4\_connect和tcp\_v4\_syn\_recv\_sock都会调用sk\_setup\_caps来设置GSO/TSO配置。

需要注意的是，只要开启了GSO，即使硬件不支持TSO，也会设置NETIF\_F\_TSO，使得sk\_can\_gso(sk)在GSO开启或者TSO开启的时候都返回true

## Jumbo Frames

以太网提出的时候是按照1500字节MTU（Maximum Transmission Unit）设计的，也就是Ethernet Frame的payload（数据段）最大是1500字节。为什么是1500字节？这是一个效率和可靠性的折中选择。因为单个包越长，效率肯定越高，但相应的丢包概率也越大。反过来，单个包越小，效率更低，因为有效数据占整个网络数据比例更低，不过相应的丢包概率也更小。 因此，IEEE802.3规定了以太网的MTU是1500。

网络传输的时候，MTU必须匹配，MTU1500向MTU9000的机器发数据没问题。但是反过来，MTU9000向MTU1500的机器发数据，因为数据太长，MTU1500的机器识别不了会丢包。因此，网络数据的收发端MTU必须匹配。另一方面，互联网从几十年前就开始构建，为了统一标准，增加兼容性，整个互联网都是根据IEEE802.3规定的MTU1500来构建。

但是，现在的网络设备可靠性有了很大的提升，可以稳定传输更大的网络包。Jumbo Frames就是MTU为9000字节的Ethernet Frames。对于Jumbo Frames来说，每个网络数据包的有效数据占比更多，因为网络协议的头部长度是固定的，网络数据包变长了只能是有效数据更多了。另一方面，以10G网络为例，MTU1500需要CPU每秒处理超过800,000个网络包，而MTU9000只需要CPU每秒处理140,000个网络包。因此，在MTU9000下，单位时间CPU需要处理的网络包更少了，留给CPU处理每个网络包的时间更多了。

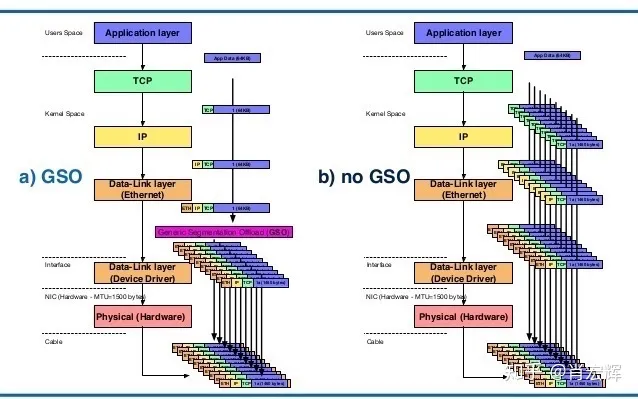
支持Jumbo Frames需要相应的硬件，最新的硬件基本都支持了，只需要简单的配置即可。但是Jumbo Frames在实际使用的时候有一定的局限性。因为Jumbo Frames提出时，互联网已经按照MTU1500搭建完了，**而MTU又必须匹配，改造全网基本不太可能。所以Jumbo Frames一般只在数据中心内部网络使用，例如内部存储网络。**连接互联网的MTU一般还是设置为1500。

## GSO（软件）

GSO全称是Generic Segmentation Offload，它只在网络数据发送时有效。GSO的作者Herbert Xu说过“If we can't use a larger MTU, we can go for the next-best thing: pretend that we're using a larger MTU.”有点像，现在我不能吃烧鸡，那老板来两片素鸡，比什么也没有强点。既然互联网是基于MTU 1500构建，在互联网上传输的网络包必须遵循MTU 1500，那如果在操作系统里面尽量晚进行IP Fragmentation，在TCP/IP协议栈里就会有一段“路径”，其上传递的网络数据是一个payload超过1500字节的网络包，相当于在传递一个大MTU的网络数据。在这段“路径”上，CPU需要处理更少的网络数据包，相应的留给CPU处理每个网络包的时间就更多了。

其实上一篇介绍的TSO也有这个思想，从用户程序到网卡之间，一直都不进行TCP Segmentation和IP Fragmentation，数据包最大可以到64K。但是，TSO只支持TCP协议，并且需要硬件网卡的支持，而GSO就是为其他场合提出。其实严格来说，除了TCP，其他的网络数据大包变小包都是发生在IP层，因此属于IP Fragmentation，所以这里叫GS(egmentation)O并不是100%恰当。

因为不依赖硬件，**又要尽可能晚的分段或者分片，所以GSO选择在发给网卡驱动的前一刻将大包分成多个小包（**如果不支持，再做GSO**）。**这样，虽然网卡收到的还是多个小的网络数据包，但是在TCP/IP协议栈里面，如下图所示，还是有一段路径，CPU需要处理少量的大包。



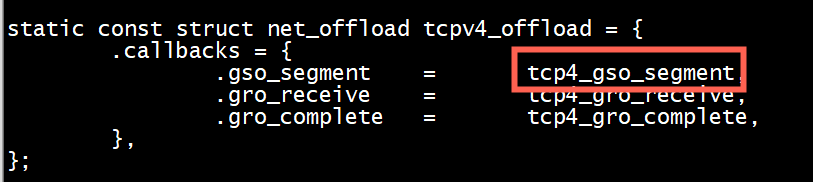
因为在发给网卡驱动的前一刻完成，所以GSO可以作为TSO的备份。在发给网卡驱动时检查网卡是否支持TSO，如果支持，将大包直接传给网卡驱动。如果不支持，再做GSO。

根据LinuxFoundation的文档，在MTU1500时，使用GSO可以使得网络吞吐量（throughput）提升17.5%。

### Udp4\_ufo\_fragment

这里是udp\_offload.c中定义的回调函数：  
static const struct net\_offload udpv4\_offload = {  
.callbacks = {  
.gso\_segment = udp4\_ufo\_fragment,  
                ...  
},  
};

### tcp4\_gso\_segment

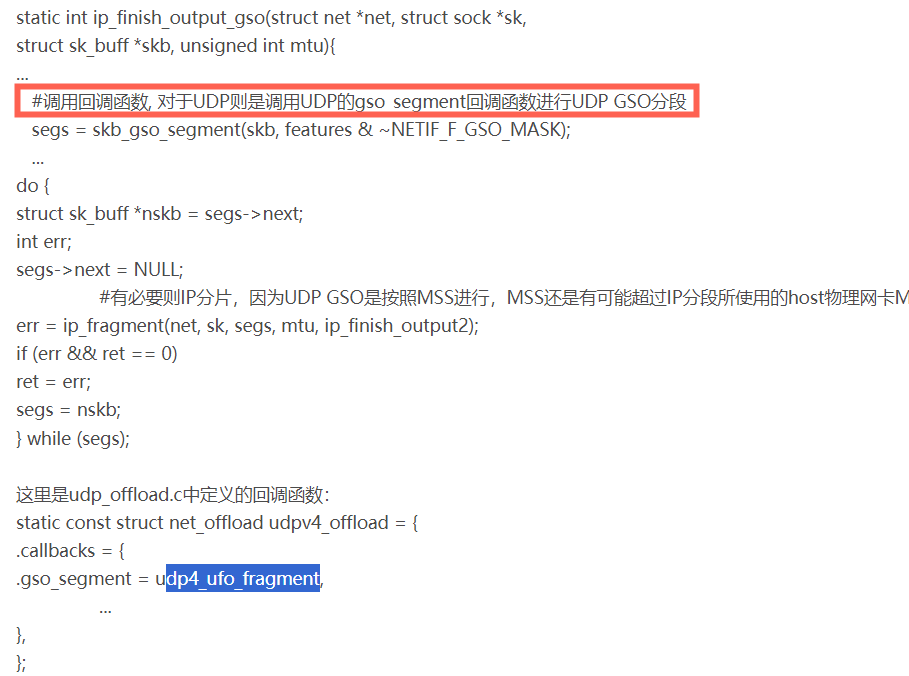


### ip\_finish\_output\_gso pk ip\_fragment

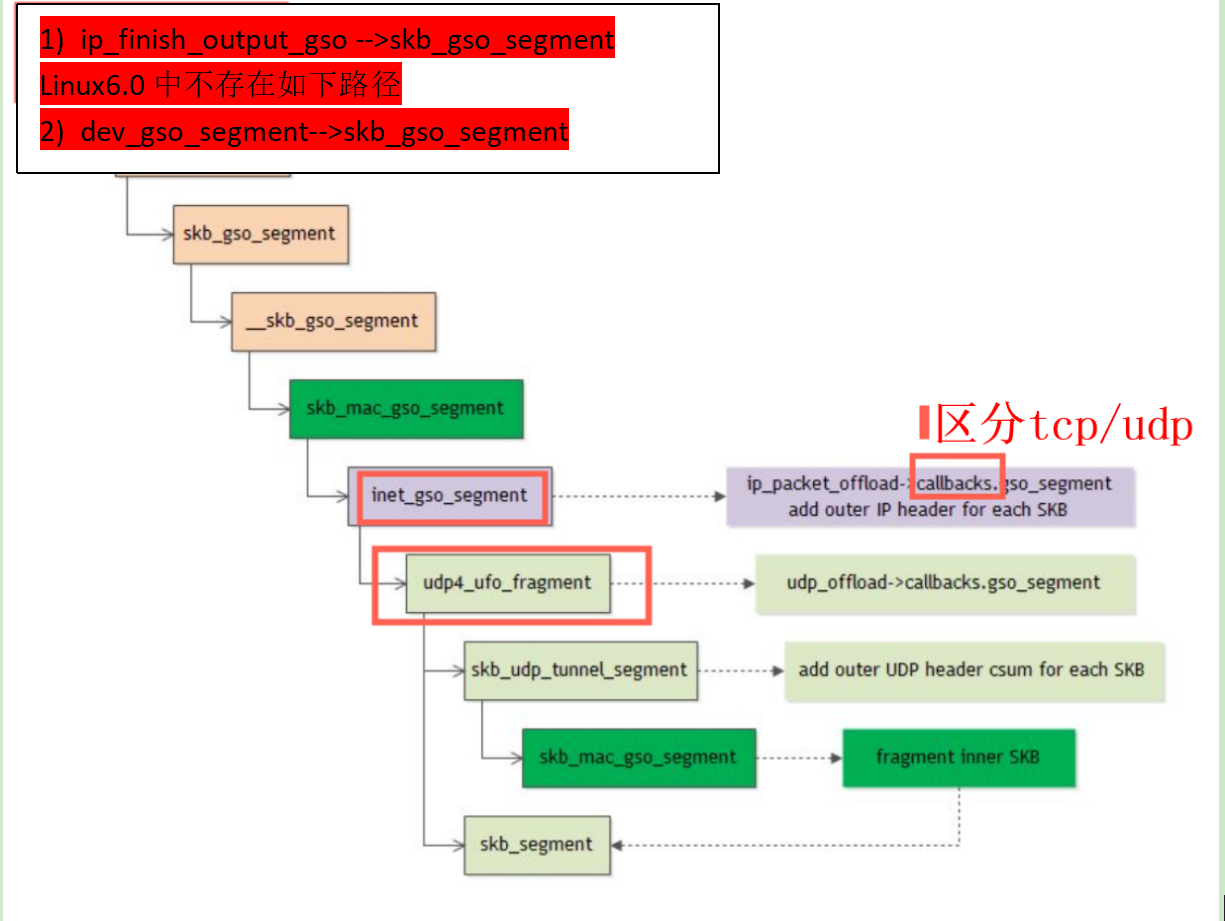
在函数 static int ip\_finish\_output\_gso(struct net \*net, struct sock \*sk,  struct sk\_buff \*skb, unsigned int mtu)

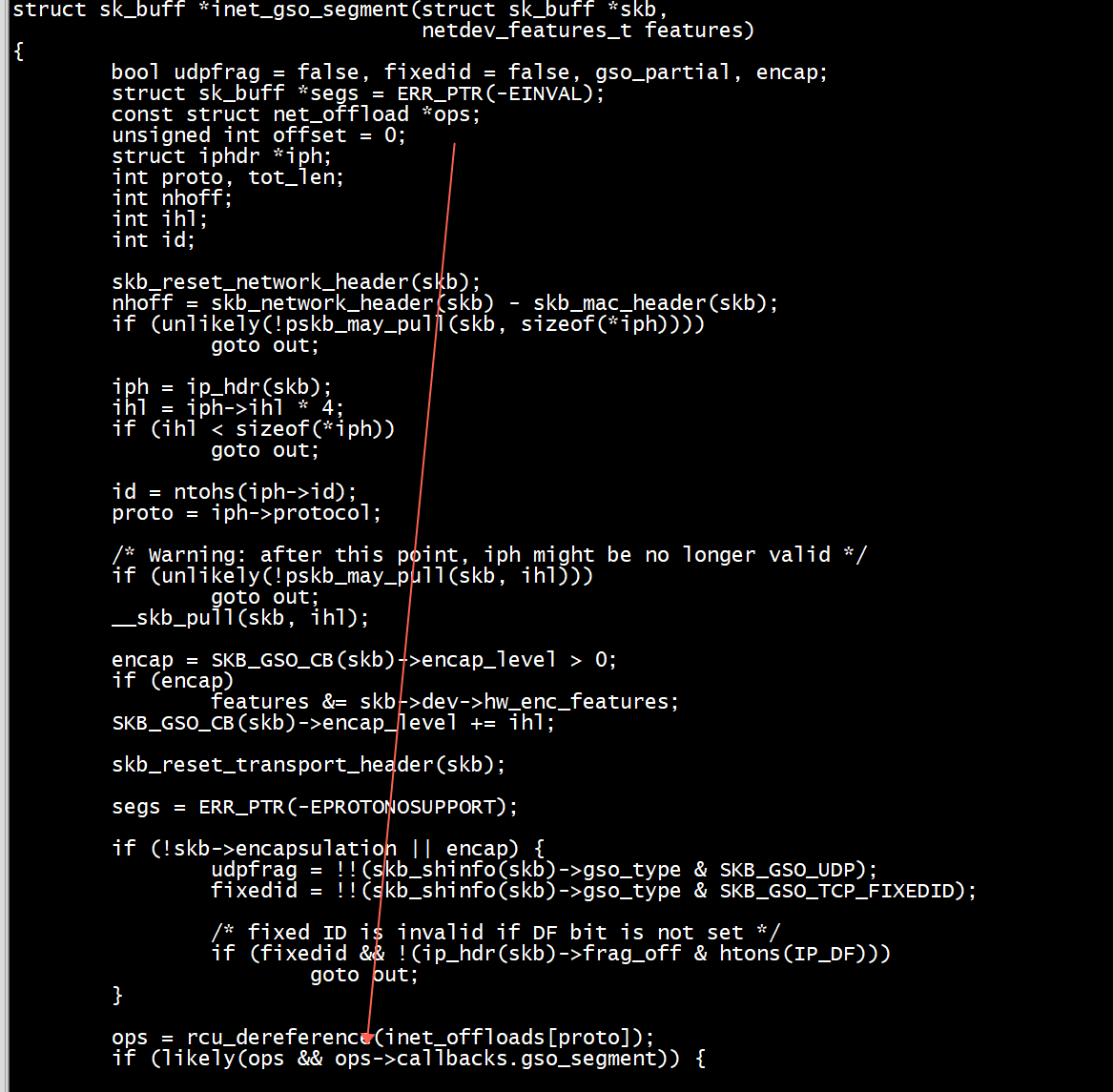
1. 首先按照 skb->MSS 做 GSO，然后再调用 ip\_fragment 做 IP 分片。
2. 可见，在通常情况下（虚机 TCP MSS 要比物理网卡 MTU 小），只做 UDP GSO 分段，IP 分片是不需要做的；只有在特殊情况下 （虚机 TCP MSS 超过了宿主机物理网卡 MTU），IP 分片才会做。这个和试验中看到的效果是相同的。





#### inet\_gso\_segment





inet\_gso\_segment函数：

1. UDP经过GSO分片后每个分片的IP头部id是一样的，这个符合IP分片的逻辑
2. 但是为什么TCP的GSO分片，IP头部的id会依次加1呢？原因是： tcp建立三次握手的过程中产生合适的mss（具体的处理机制参见TCP/IP详解P257），这个mss肯定是<= 网络层的最大路径MTU，然后tcp数据封装成ip数据包通过网络层发送，当服务器端传输层接收到tcp数据之后进行tcp重组。所以正常情况下tcp产生的ip数据包在传输过程中是不会发生分片的！由于GSO应该保证对外透明，所以其效果应该也和在TCP层直接分片的效果是一样的，所以这里对UDP的处理是IP分片逻辑，但对TCP的处理是构造**新的**skb逻辑。

**小结：对于GSO**

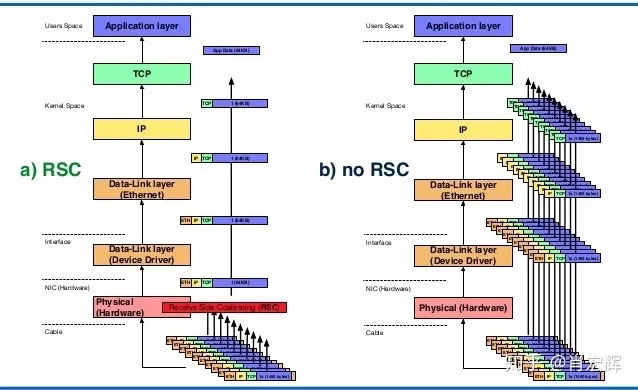
1. UDP:所有分片ip头部id都相同，设置IP\_MF分片标志(除最后一片) （等同于IP分片）
2. TCP:分片后，每个分片IP头部中id加1, （等同于TCP分段,构造新的skb）

### skb\_is\_gso



## LRO（废除）

LRO全称是Large Receive Offload，或者又称为RSC（Receive Side Coalescing），从名字可以看出它只在网络数据接收时有效。LRO是TSO的逆方向实现，是指网卡将同一个TCP连接的TCP Segments 合并成一个大的TCP包，再传给操作系统。这样避免了操作系统处理并合并多个小包，减少了CPU的运算时间，并且在TCP/IP协议栈，CPU需要处理更少的网络数据包。与TSO一样，LRO也需要网卡的支持。



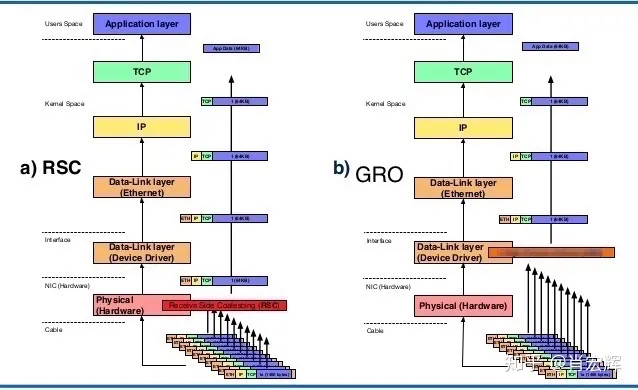
但是与TSO不一样的是，LRO并没那么好用。因为TSO发生在数据的发送方，发送方掌握了网络数据的全部信息，发送方可以按照自己的判断控制发送的流程。而LRO发生在数据的接收方，而且是相对于数据发送方的异步接收，所以LRO只能基于当前获取到的有限数据和信息做出合并，存在一定的困难。这就像我们拆一个东西很容易，但是要重新组装回去很难一样。

LRO可能会丢失重要的数据，例如数据发送方在Header加了一些字段来区分不同的网络包。合并可能导致这些字段的丢失，因为合并之后只有一个Header了。而且当操作系统需要转发数据时，合并之后的网络包可能需要重新被分段/片。再重新分成小包，原来Header里面的差异字段就彻底丢失了。因为LRO的局限性，在一些最新的网卡上，LRO已经被删除了。

## GRO（软件）

GRO全称是Generic Receive Offload，这是GSO在接收端的对应。GRO的作者与GSO是同一个人，都是Herbert Xu。不像GSO作为TSO的替补，GRO逐渐取代了LRO。因为GRO运行在系统内核（软件实现），掌握的信息更多，GRO可以用更加严格的规则来合并网络数据包。因为合并的时候更严格，所以可以避免关键的信息丢失。另一方面，在一些需要转发的场合，GRO可以利用GSO的代码来重新分段。

其他的优点还有，GRO也更加通用，不仅不依赖硬件设备，还支持TCP协议以外的协议。



## UFO（废除）

UFO全称是UDP fragmentation offload。从名字可以看出，这是针对UDP的优化。但是不像TCP，UDP没有Segmentation的过程，用户程序发给UDP多长的数据（当然要控制在64K以内），UDP都会转给IP层。IP层会根据MTU进行Fragmentation。UFO使得网络设备，例如网卡，可以将一个超长的UDP数据段（超过MTU），分成多个IPv4分片（fragment）。因为在网卡做了，所以，CPU的运算量被节省下来了。

不过，在最新的linux内核中，UFO已经被弃用了。因为除了TSO，其他的offload基本上都是在IP层做Fragmentation，那UFO也没有必要单独存在，因此它与GSO合并表示了。

## TSO和GSO区别

1, TSO只有第一个分片有TCP头和IP头，接着的分段只有IP头。意味着第一个分段丢失，所有分段得重传。  
2, GSO在分段时会调用TCP或UDP的回调函数(udp4\_ufo\_fragment)为每个分段都加上IP头,l4header，由于分段是**通过mss设置的（mss由发送端设置），所以长度仍然可能超过mtu值，所以在IP层还得再分片(代码位于dev\_hard\_start\_xmit)。**

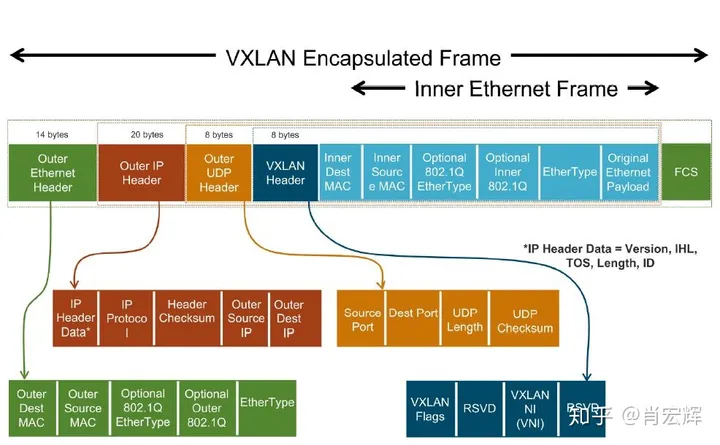
### 发数据

TSO(TCP分段，TCP Segmentation Offload)在TCP处做，UFO(UDP分片，UDP Fragmentation Offload）因为UDP不支持分段所以移到下层的IP层做分片。TSO是使得网络协议栈能够将大块buffer推送至网卡，然后网卡执行分片工作，这样减轻了CPU 的负荷，但TSO需要硬件来实现分片功能；

GSO(Generic Segmentation Offload)比TSO更通用，基本思想就是尽可能的推迟数据分片直至发送到网卡驱动之前，如果硬件不支持TSO分段则由dev\_hard\_start\_xmit中的dev\_gso\_segment先软件分段的segs赋值给skb->next（skb->next = segs）, 如果网卡硬件支持分段则直接将GSO大帧（skb-next)传给网卡驱动(dev\_hard\_start\_xmit中的ndo\_start\_xmit)（所以它传给virtio驱动的是GSO大帧），然后继续进行IP分片后再发往网卡。GSO自动检测网卡支持特性， 硬件不支持也可以使用GSO它更通用（TSO一定需要硬件支持）。当打开GSO时，GSO会在xmit那块做GSO分片时调用TCP/UDP的回调函数自动添加TCP/UDP头（不使用GSO的只有第一个分片有TCP/UDP头，后面接着的分片是没有的，这也是为什么在虚机里打开TSO/GSO时对于隧道会多出一块数据的原因），然后再调IP层回调函数为每个分片添加IP头。

## tx-udp\_tnl-segmentation

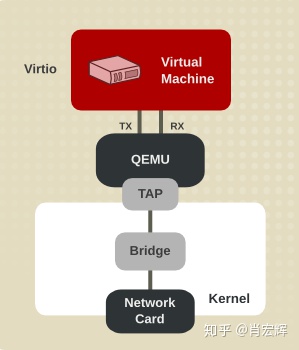
Overlay网络，例如VxLAN，现在应用的越来越多。Overlay网络可以使得用户不受物理网络的限制，进而创建，配置并管理所需要的虚拟网络连接。同时Overlay可以让多个租户共用一个物理网络，提高网络的利用率。Overlay网络有很多种，但是最具有代表性的是VxLAN。VxLAN是一个MAC in UDP的设计，具体格式如下所示。



从VxLAN的格式可以看出，以VxLAN为代表的Overlay网络在性能上存在两个问题。一个是Overhead的增加，VxLAN在原始的Ethernet Frame上再包了一层Ethernet+IP+UDP+VXLAN，这样每个Ethernet Frame比原来要多传输50个字节。所以可以预见的是，Overlay网络的效率必然要低于Underlay网络。另一个问题比传50个字节更为严重，那就是需要处理这额外的50个字节。这50个字节包括了4个Header，每个Header都涉及到拷贝，计算，都需要消耗CPU。而我们现在迫切的问题在于CPU可以用来处理每个网络数据包的时间更少了。

首先，VxLAN的这50个字节是没法避免的。其次，那就只能降低它的影响。这里仍然可以采用Jumbo Frames的思想，因为50个字节是固定的，那网络数据包越大，50字节带来的影响就相对越小。

先来看一下虚拟机的网络连接图。虚拟机通过QEMU连接到位于宿主机的TAP设备，之后再通过虚机交换机转到VTEP（VxLAN Tunnel EndPoint），封装VxLAN格式，发给宿主机网卡。



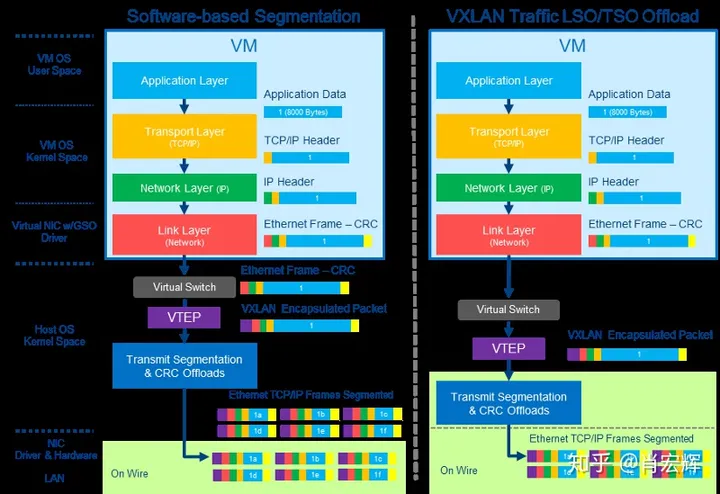
理想情况就是，一大段VxLAN数据直接传给网卡，由网卡去完成剩下的分片，分段，并对分成的小的网络包分别封装VxLAN，计算校验和等工作。这样VxLAN对虚机网络带来影响就可以降到最低。实际中，这是可能的，但是需要一系列的前提条件。

首先，虚拟机要把大的网络包发到宿主机。因为虚拟机里面也运行了一个操作系统，也有自己的TCP/IP协议栈，所以虚拟机完全有能力自己就把大的网络包分成多个小的网络包。从前面介绍的内容看，只有TSO才能真正将一个大的网络包发到网卡。GSO在发到网卡的时候，已经在进入驱动的前一刻将大的网络包分成了若干个小的网络数据包。所以这里要求：虚机的网卡支持TSO（Virtio默认支持），并且打开TSO（默认打开），同时虚机发出的是TCP数据。

之后，经过QEMU，虚拟交换机的转发，VTEP的封装，这个大的TCP数据被封装成了VxLAN格式。50个字节的VxLAN数据被加到了这个大的TCP数据上。接下来问题来了，这本来是个TCP数据，但是因为做了VxLAN的封装，现在看起来像是个UDP的数据。如果操作系统不做任何处理，按照前面的介绍，那就应该走GSO做IP Fragmentation，并在发送给网卡的前一刻分成多个小包。这样，如果网卡本来支持TSO现在就用不上了。并且更加严重的是，现在还没做TCP Segmentation。我们在上一篇花了很大的篇幅介绍其必要性的TCP Segmentation在这里也丢失了。

对于现代的网卡，除了TSO，GSO等offload选项外，还多了一个选项tx-udp\_tnl-segmentation。如果这个选项打开，操作系统自己会识别封装成VxLAN的UDP数据是一个tunnel数据，并且操作系统会直接把这一大段VxLAN数据丢给网卡去处理。在网卡里面，网卡会针对内层的TCP数据，完成TCP Segmentation。之后再为每个TCP Segment加上VxLAN封装（50字节），如下图右所示。这样，VxLAN封装对于虚拟机网络来说，影响降到了最低。

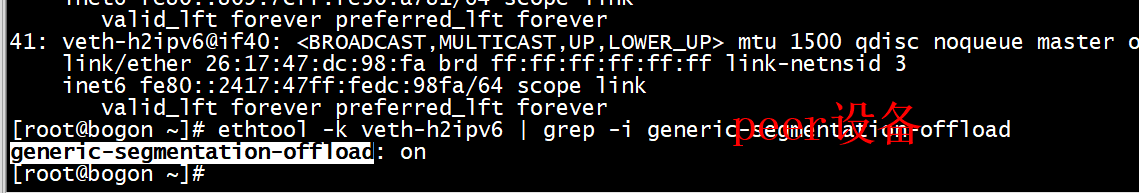
从前面描述看，要达成上述的效果，需要宿主机网卡同时支持TSO和tx-udp\_tnl-segmentation。如果这两者任意一个不支持或者都不支持。那么系统内核会调用GSO，将封装成VxLAN格式的大段TCP数据，在发给网卡驱动前完成TCP Segmentation，并且为每个TCP Segment加上VxLAN封装。如下图左所示。

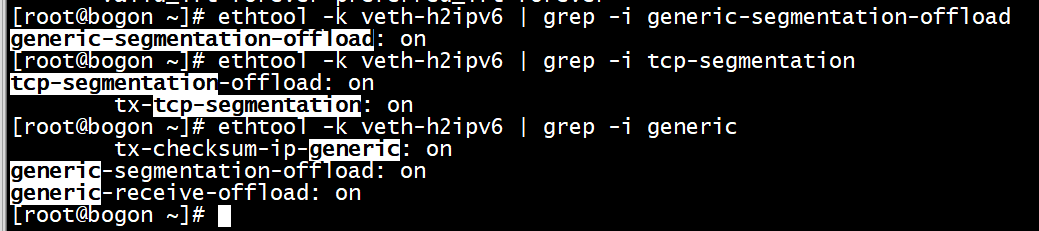


如果关闭虚拟机内的TSO，或者虚拟机内发送的是UDP数据。那么在虚拟机的TCP/IP协议栈会调用GSO，发给虚拟机网卡驱动的前一刻，完成了分段、分片。虚拟机最终发到QEMU的网络数据包就是多个小的网络数据包。这个时候，无论宿主机怎么配置，都需要处理多个小的网络包，并对他们做VxLAN封装。

### 虚拟机gso/tso特性

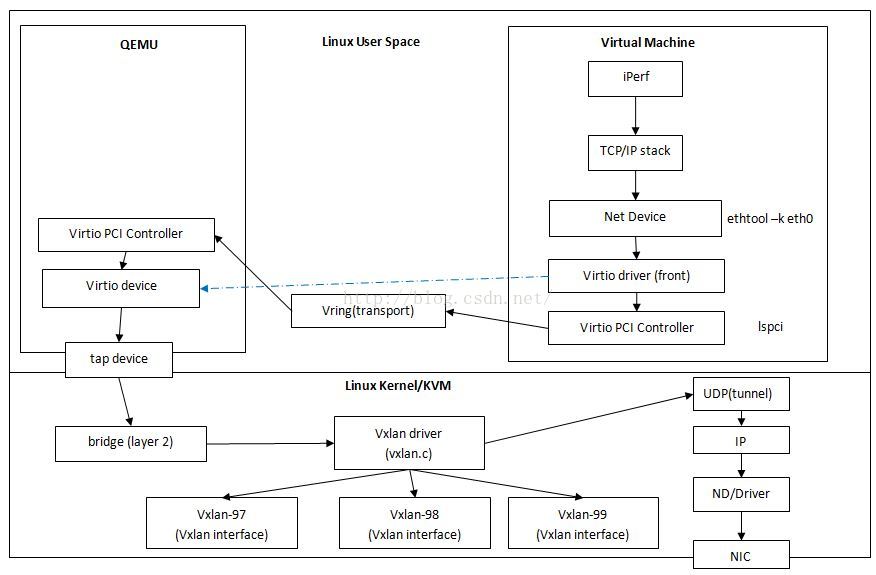
Peer设备的gso、tso都是打开的



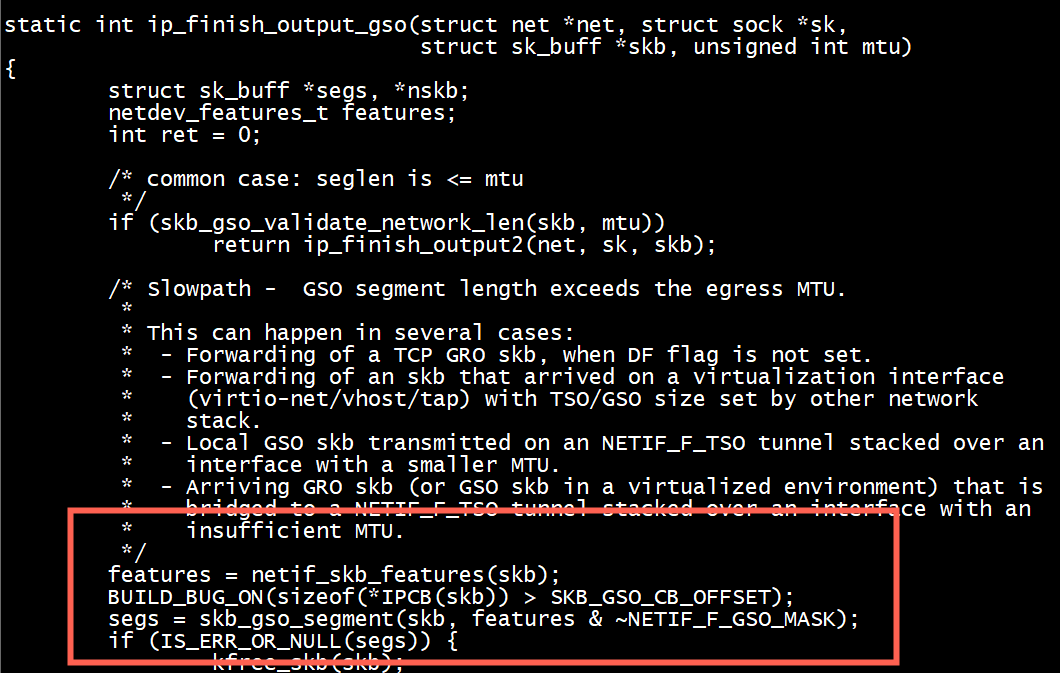


### vxlan的实现

从guest出来的tcp数据到达host的vxlan driver时会调用vxlan\_xmit, 它的主要逻辑是获取 vxlan dev，然后为 sk\_buff 中的每一个skb调用vxlan\_xmit\_skb方法, vxlan\_xmit\_skb除了计算 tos，ttl，df，src\_port，dst\_port，md，flags等以外，也调用skb\_set\_inner\_protocol(skb, htons(ETH\_P\_TEB))设置GSO参数。最后调用udp\_tunnel\_xmit\_skb将skb传给udp tunnel协议栈继续处理。



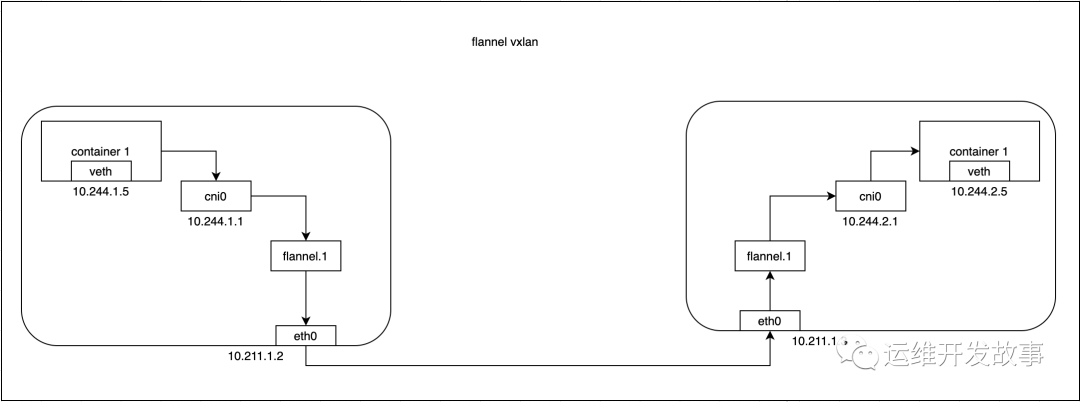
1. netif\_needs\_gso先判断网卡是否支持GSO（guest里的virtio nic肯定支持），if(unlikely(foo))认为foo通常为0， 所以如果网卡支持TSO时dev\_gso\_segment将返回0（skb->next==NULL)，也就是说，当guest网卡的GSO特性打开时，guest会直接将没有分段的GSO大帧传递到virtio-net driver中。



（2）virtio\_net.c中的如下代码定义了virtio-net driver的xmit函数为start\_xmit，它会直接将这个GSO大帧通过vring(<=64K)传给host上的virtio backend driver.  
static const struct net\_device\_ops virtnet\_netdev = {  
        ...  
.ndo\_start\_xmit      = start\_xmit,  
  
（3）然后host上的tap和bridge都会原封不动动转发这个GSO大帧(payload, inner-tcp, inner-ip, inner-ethernet)。  
  
（4）host上的vxlan driver在原GSO大帧上添加vxlan帧头，从(payload, inner-tcp, inner-ip, inner-ethernet)变成(payload, inner-tcp, inner-ip, inner-ethernet, vxlan)。  
  
（5）如果host不支持GSO，那么在host的IP层（ip fragmen）直接对外层的UDP分片，注意：只有第1个分片有UDP头，接下来的分片没有UDP头。  
   分片1： payload, inner-tcp, inner-ip, inner-ethernet, vxlan，outer-udp, outer-ip  
   分片2： payload, inner-tcp, inner-ip, inner-ethernet, vxlan，outer-ip  
  
（6）如果host支持GSO，如果物理网卡支持TSO直接将大帧(payload, inner-tcp, inner-ip, inner-ethernet, vxlan)将给物理网卡的TSO去硬件分片。如果物理网卡不支持TSO将调用skb\_gso\_segment软件执行GSO。GSO由于会调用UDP的回调函数，但vxlan没有GSO回调函数，所以这里的GSO分片应该每一个分片都有UDP头部，但是只有第一个分片有vxlan头部。  
   分片1： payload, inner-tcp, inner-ip, inner-ethernet, vxlan，outer-udp, outer-ip  
   分片2： payload, inner-tcp, inner-ip, inner-ethernet，outer-udp, outer-ip

### Vxlan mtu

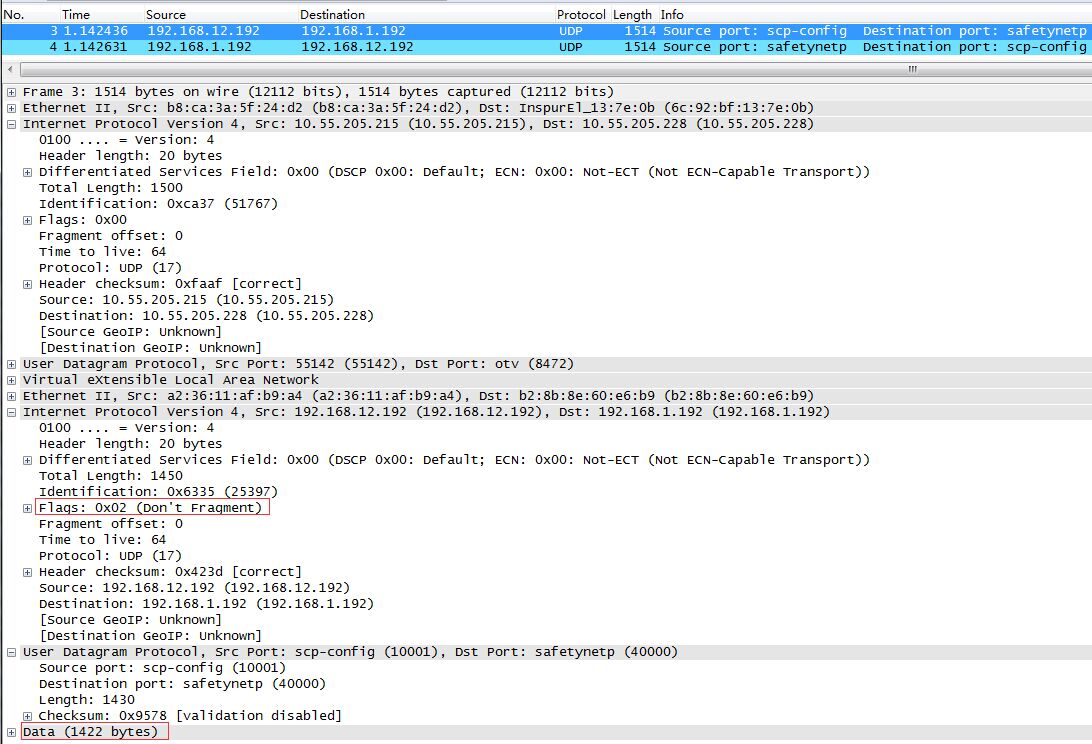
不同节点上的pod要互相进行访问时，需要通过主机路由，需要经过内核的封包解包操作，整个过程如下所示：

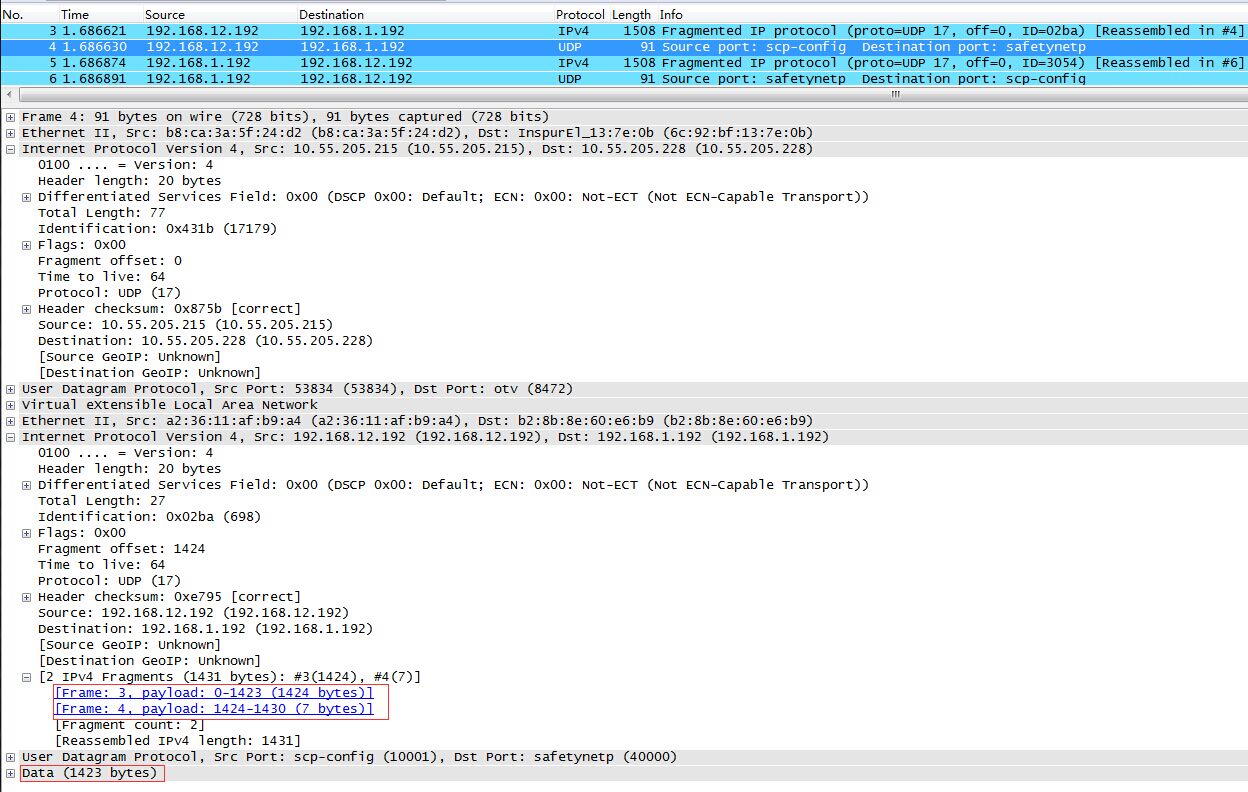




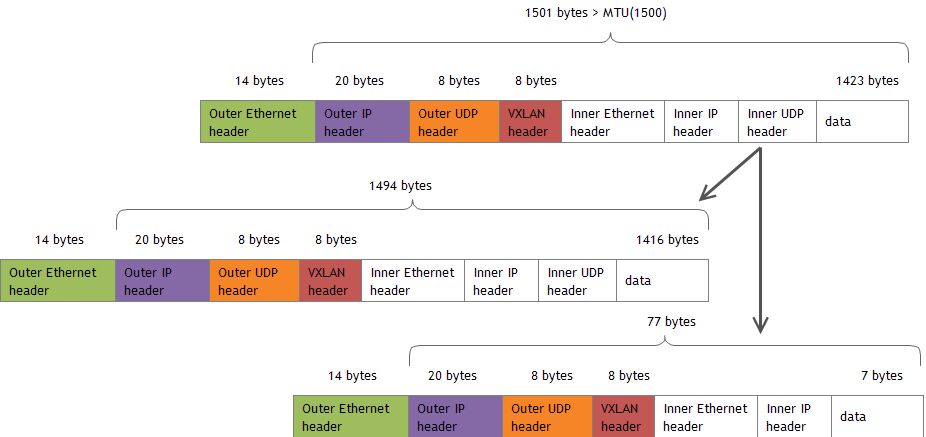
### VXLAN example

对于VXLAN，有50字节的额外开销：Outer L2 header(14 bytes) + Outer L3 header(20 bytes) + Outer UDP header(8 bytes) + VXLAN header(8 bytes)。如果underlay network的MTU为1500，需要将VXLAN设备的MTU设置为1450。

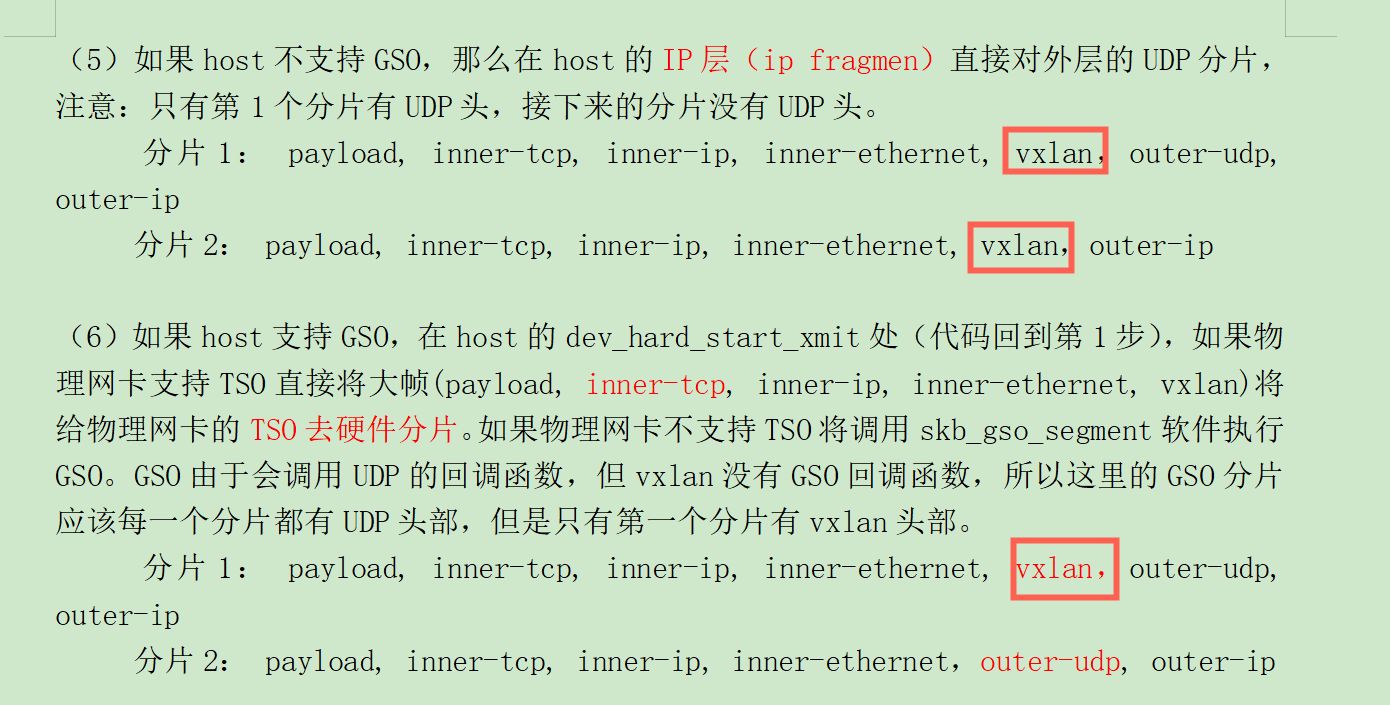
如果基于VXLAN发送一个1422字节（1450 - 20 - 8)的UDP包，内层UDP包不会发生分片： [](https://github.com/hustcat/hustcat.github.io/blob/master/assets/ip_fragment/vxlan_1422.jpg)

如果发送一个1423字节的UDP包，我们就会观察内层UDP包发生分片： [](https://github.com/hustcat/hustcat.github.io/blob/master/assets/ip_fragment/vxlan_1423.jpg)

\*\*\* 值得注意的是，第1个frame只有1508字节，也就是说只包含1416字节的用户数据，第2个frame包含剩下的7个字节。（原因呢？？？） \*\*\*

[](https://github.com/hustcat/hustcat.github.io/blob/master/assets/ip_fragment/vxlan_fragment.jpg)

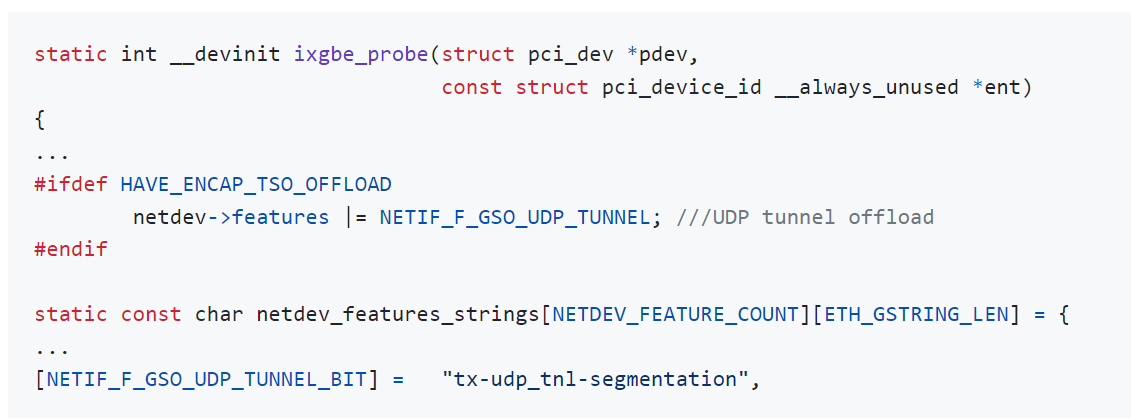
第二个报文也有vxlan



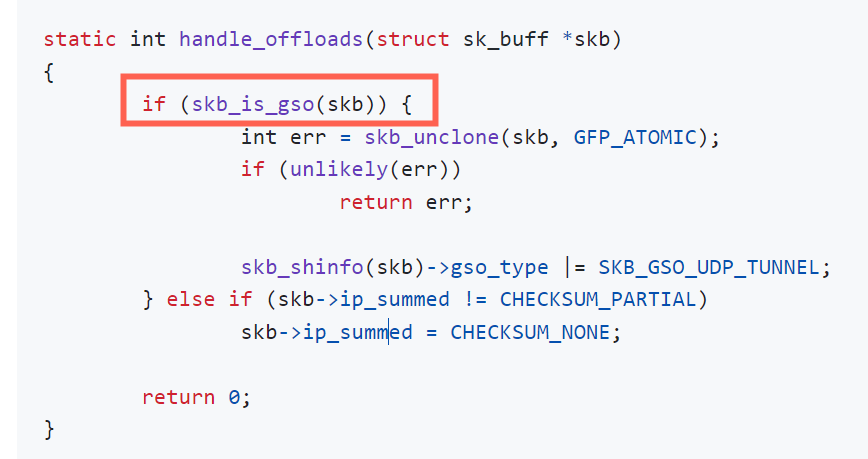
### VXLAN hardware offload

ethtool -k eno1 | grep tx-udp\_tnl-segmentation

tx-udp\_tnl-segmentation: on

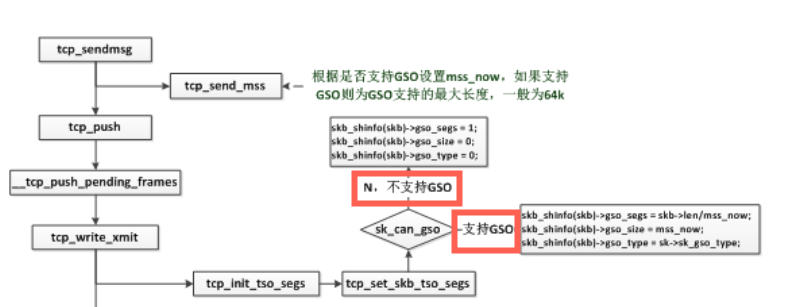


VXLAN设备在发送数据时，会设置SKB\_GSO\_UDP\_TUNNEL:



值得注意的是，该特性只有当内层的packet为TCP协议时，才有意义。前面已经讨论ixgbe不支持UFO，所以对UDP packet，最终会在推送给物理网卡时(dev\_hard\_start\_xmit)进行软件GSO。

# ixgbe\_tso



1. ixgbe\_xmit\_frame\_ring-->ixgbe\_tso
2. tcp\_set\_skb\_tso\_segs会设置gso\_size,决定是进入ixgbe\_tso处理

