Segmentation Offloads

Introduction

This document describes a set of techniques in the Linux networking stack to take advantage of segmentation offload capabilities of various NICs.

The following technologies are described:

- TCP Segmentation Offload - TSO
- UDP Fragmentation Offload - UFO
- IPIP, SIT, GRE, and UDP Tunnel Offloads
- Generic Segmentation Offload - GSO
- Generic Receive Offload - GRO
- Partial Generic Segmentation Offload - GSO_PARTIAL
- SCTP acceleration with GSO - GSO_BY_FRAGS

TCP Segmentation Offload

TCP segmentation allows a device to segment a single frame into multiple frames with a data payload size specified in skb_shinfo()->gso_size. When TCP segmentation requested the bit for either SKB_GSO_TCPV4 or SKB_GSO_TCPV6 should be set in skb_shinfo()->gso_type and skb_shinfo()->gso_size should be set to a non-zero value.

TCP segmentation is dependent on support for the use of partial checksum offload. For this reason TSO is normally disabled if the Tx checksum offload for a given device is disabled.

In order to support TCP segmentation offload it is necessary to populate the network and transport header offsets of the skbuff so that the device drivers will be able determine the offsets of the IP or IPv6 header and the TCP header. In addition as CHECKSUM_PARTIAL is required csum_start should also point to the TCP header of the packet.

For IPv4 segmentation we support one of two types in terms of the IP ID. The default behavior is to increment the IP ID with every segment. If the GSO type SKB_GSO_TCP_FIXEDID is specified then we will not increment the IP ID and all segments will use the same IP ID. If a device has NETIF_F_TSO_MANGLEID set then the IP ID can be ignored when performing TSO and we will either increment the IP ID for all frames, or leave it at a static value based on driver preference.

UDP Fragmentation Offload

UDP fragmentation offload allows a device to fragment an oversized UDP datagram into multiple IPv4 fragments. Many of the requirements for UDP fragmentation offload are the same as TSO. However the IPv4 ID for fragments should not increment as a single IPv4 datagram is fragmented.

UFO is deprecated: modern kernels will no longer generate UFO skbs, but can still receive them from tuntap and similar devices. Offload of UDP-based tunnel protocols is still supported.

IPIP, SIT, GRE, UDP Tunnel, and Remote Checksum Offloads

In addition to the offloads described above it is possible for a frame to contain additional headers such as an outer tunnel. In order to account for such instances an additional set of segmentation offload types were introduced including SKB_GSO_IPXIP4, SKB_GSO_IPXIP6, SKB_GSO_GRE, and SKB_GSO_UDP_TUNNEL. These extra segmentation types are used to identify cases where there are more than just 1 set of headers. For example in the case of IPIP and SIT we should have the network and transport headers moved from the standard list of headers to "inner" header offsets.

Currently only two levels of headers are supported. The convention is to refer to the tunnel headers as the outer headers, while the encapsulated data is normally referred to as the inner headers. Below is the list of calls to access the given headers:

IPIP/SIT Tunnel:

```
            Outer                   Inner
MAC         skb_mac_header
Network     skb_network_header      skb_inner_network_header
Transport   skb_transport_header
```

UDP/GRE Tunnel:

```
            Outer                   Inner
MAC         skb_mac_header          skb_inner_mac_header
Network     skb_network_header      skb_inner_network_header
Transport   skb_transport_header    skb_inner_transport_header
```

In addition to the above tunnel types there are also SKB_GSO_GRE_CSUM and SKB_GSO_UDP_TUNNEL_CSUM. These two additional tunnel types reflect the fact that the outer header also requests to have a non-zero checksum included in the outer

header.

Finally there is SKB_GSO_TUNNEL_REMCSUM which indicates that a given tunnel header has requested a remote checksum offload. In this case the inner headers will be left with a partial checksum and only the outer header checksum will be computed.

Generic Segmentation Offload

Generic segmentation offload is a pure software offload that is meant to deal with cases where device drivers cannot perform the offloads described above. What occurs in GSO is that a given skbuff will have its data broken out over multiple skbuffs that have been resized to match the MSS provided via skb_shinfo()->gso_size.

Before enabling any hardware segmentation offload a corresponding software offload is required in GSO. Otherwise it becomes possible for a frame to be re-routed between devices and end up being unable to be transmitted.

Generic Receive Offload

Generic receive offload is the complement to GSO. Ideally any frame assembled by GRO should be segmented to create an identical sequence of frames using GSO, and any sequence of frames segmented by GSO should be able to be reassembled back to the original by GRO. The only exception to this is IPv4 ID in the case that the DF bit is set for a given IP header. If the value of the IPv4 ID is not sequentially incrementing it will be altered so that it is when a frame assembled via GRO is segmented via GSO.

Partial Generic Segmentation Offload

Partial generic segmentation offload is a hybrid between TSO and GSO. What it effectively does is take advantage of certain traits of TCP and tunnels so that instead of having to rewrite the packet headers for each segment only the inner-most transport header and possibly the outer-most network header need to be updated. This allows devices that do not support tunnel offloads or tunnel offloads with checksum to still make use of segmentation.

With the partial offload what occurs is that all headers excluding the inner transport header are updated such that they will contain the correct values for if the header was simply duplicated. The one exception to this is the outer IPv4 ID field. It is up to the device drivers to guarantee that the IPv4 ID field is incremented in the case that a given header does not have the DF bit set.

SCTP acceleration with GSO

SCTP - despite the lack of hardware support - can still take advantage of GSO to pass one large packet through the network stack, rather than multiple small packets.

This requires a different approach to other offloads, as SCTP packets cannot be just segmented to (P)MTU. Rather, the chunks must be contained in IP segments, padding respected. So unlike regular GSO, SCTP can't just generate a big skb, set gso_size to the fragmentation point and deliver it to IP layer.

Instead, the SCTP protocol layer builds an skb with the segments correctly padded and stored as chained skbs, and skb_segment() splits based on those. To signal this, gso_size is set to the special value GSO_BY_FRAGS.

Therefore, any code in the core networking stack must be aware of the possibility that gso_size will be GSO_BY_FRAGS and handle that case appropriately.

There are some helpers to make this easier:

- skb_is_gso(skb) && skb_is_gso_sctp(skb) is the best way to see if an skb is an SCTP GSO skb.
- For size checks, the skb_gso_validate_*_len family of helpers correctly considers GSO_BY_FRAGS.
- For manipulating packets, skb_increase_gso_size and skb_decrease_gso_size will check for GSO_BY_FRAGS and WARN if asked to manipulate these skbs.

This also affects drivers with the NETIF_F_FRAGLIST & NETIF_F_GSO_SCTP bits set. Note also that NETIF_F_GSO_SCTP is included in NETIF_F_GSO_SOFTWARE.