BIG TCP

Eric Dumazet & Coco Li @ Google

Netdev 0x15, July 2021

Overall plan/roadmap

- Problem statement (tldr: TCP is slooooow, use BIG packets)

- Prototype patch to have estimates of cpu savings, and feasibility.
- Changes in IPv6 stack.
- Change in TX path (dev->gso_ipv6_max_size)
- Change in GSO stack.
- Change in GRO stack.
- Changes in skb layout (optional)
- Changes in drivers for TX.

TCP in Linux, the legacy of skbs

Traditionally, transport protocols in Linux have used skbs to hold packets for both TX and RX.

skbs are considered fat/complex objects, because they contain metadata for all possible networking features that linux supports. We fight hard to limit their wild expansion.

Yet, skbs are pretty limited in term of payload density, even with large MTU.

On both TX and RX, TSO/GSO packets and GRO packets are limited to ~64KB of payload.

GSO and GRO came late

At first, skbs would contain only one linear segment (one MSS in TCP terms) at a time.

Then Scatter Gather (SG) was added to be able to split header/data in multiple area.

Then GSO/GRO has been added as NIC vendors started to support segmentation offload (TSO) and receive aggregation (LRO).

GSO/GRO were designed as software fallbacks and optimizations.

struct skb_shared_info frags[MAX_SKB_FRAGS]

GSO/GRO have been designed with practical limits of available NICs and IPv4 max packet length.

When GRO completes a packet aggregation on IPv4, iph->tot_len is updated to reflect ‘the size’ of the packet as in:

(size of one set of IP/TCP header) + size(payload)

Because iph->tot_len is a 16bit field, this limits GRO to ~64KB sizes.
Because GSO is symmetric to GRO, it has the same limit.
And of course, IPv6 has the same limits.

64KB per skb became too small

Time to transmit a ~64KB ‘packet’

1Gbit : 500 usec

10Gbit : 50 usec

100Gbit : 5 usec

200Gbit : 2.5 usec

400Gbit: 1.25 usec

Can we make bigger batches ?

GSO/GRO are essential to good performance, because they added device offload (if NIC supports TSO), but also cut number of packets sent/received through all networking stack (IP, qdisc, device layer)

While RX could be tweaked to receive a _list_ of skbs (as added in core stack around 4.19 by Edward Cree), TX would still have issues:
- There is no support yet for list of skbs at TX side.
- when TCP pacing is used (FQ packet scheduler), each ~64KB GSO packet would still have to be delivered few usec after prior one and would still require multiple interrupts per GSO in common cases.

RFC 2675 to the rescue (???)

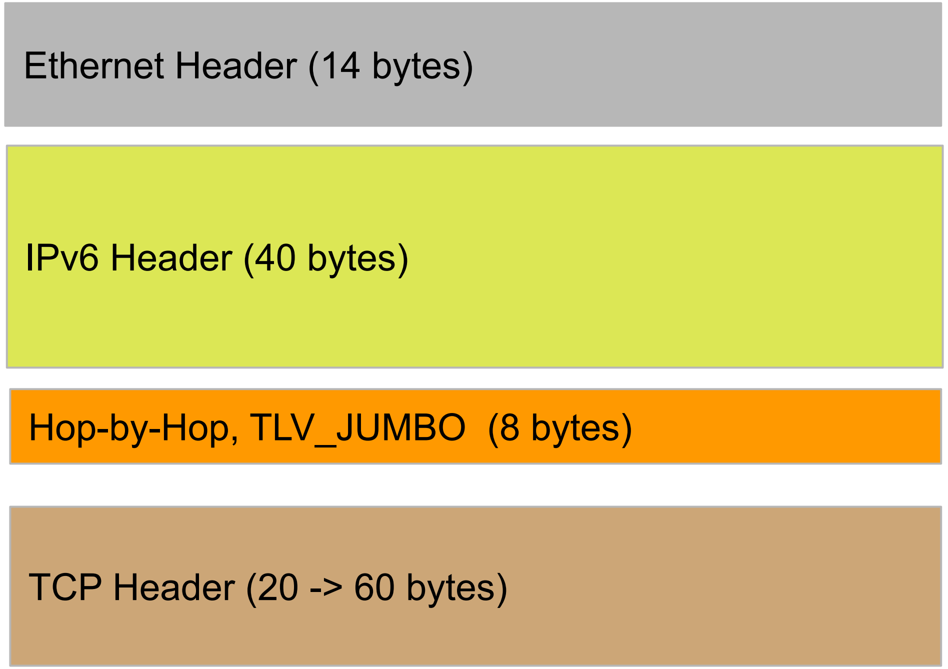
This 22 years old RFC describes IPv6 jumbograms.

In short, packets bigger than 64KB are supported, with the addition of one extension header (Hop-By-Hop, with a jumbo TLV), total of 8 bytes.

AFAIK, IPv6 jumbograms are not widely used on the wires, mainly because MTU are rarely bigger than 64KB.

But we could use them internally, to allow much bigger TCP logical packets, and thus reduce number of interactions, for IPv6 traffic.

Where is this extension header


Ethernet Header (14 bytes)
IPv6 Header (40 bytes)
Hop-by-Hop, TLV_JUMBO (8 bytes)
TCP Header (20 -> 60 bytes)

The payload_len field is 16bit ‘only’, value must be zero

The payload_len is using a 32bit field

Ugly jumbo...prototype patch

I took a mlx4 NIC, and fed it with big TCP packets (hacking GSO_MAX_SIZE)
I added a hack into skb_gro_receive() to allow bigger GRO packets.
I added a hack into ip6_rcv_core() to not trim skb based on ip6hdr->payload_len (Because GRO is not yet adding an exthdr)

Basically, I did not add IPv6 exthdr with TLV_JUMBO yet, I made sure that at least one common NIC was able to send bigger TSO packets, and that overall performance was much better. Note that I was using TCP MSS of ~4096 bytes, so that each incoming packet was consuming exactly one order-0 page.

Ugly jumbo…prototype patch

This worked, with reduced system load.

We got strange artifacts with tcpdump, who was confused by ‘strange’ zero payload_len in IPv6 header or wrapped payload_len for big GRO packets.

This encouraging result after few hours of work was enough for us to start working on ‘real patches’, with Coco Li (<lixiaoyan@google.com>)

IPv6 state of the art vs Jumbograms (RX)

At RX side, *ipv6_parse_hopopts*() is able to dissect a HOP-by-HOP extension header with a valid IPV6_TLV_JUMBO TLV just fine.
ipv6_hop_jumbo() does implement all checks requested by RFC 2675

One minor improvement can be done to avoid the cost of indirect calls done from ip6_parse_tlv(), using helpers from <linux/indirect_call_wrapper.h>

IPv6 state of the art vs Jumbograms (TX)

At TX side, *__ip6_local_out*() has the following part which looks pretty strange:

```
len = skb->len - sizeof(struct ipv6hdr);
if (len > IPV6_MAXPLEN)
        len = 0;
ipv6_hdr(skb)->payload_len = htons(len);
```

At least for observability [1] purposes, we probably want here to insert a real extension header with proper JUMBO TLV

[1] Hopefully tcpdump is able to dissect/skip it.

Changes in TX path

We want to back dev->gso_max_size with a new **dev->gso_ipv6_max_size** so that NIC drivers supporting bigger TSO packets can advertise the capability. Default value will mirror dev->gso_max_size

team/bonding drivers would propagate the min value of the members of the aggregate.

Note: Some NIC are able to send ~256KB TSO packets, but without the IPv6 extension header (Hop-by-HOP + TLV_JUMBO). They might have to be modified to remove/skip the extra header added in *__ip6_local_out*().

Change in GSO stack

At least for forwarding workloads, we might need to be able to transform a Big TSO packets (more than 64KB) into a series of smaller TSO packets, meeting the egress device gso_ipv6_max_size

This problem is orthogonal to JUMBO support, since GRO started to use frag_list, to be able to cook GRO packets with 44 or 45 segments (~64KB packets with MTU 1500) in linux-3.13

8a29111c7ca6 net: gro: allow to build full sized skb

Changes in GRO stack

skb_gro_receive() has an hardcoded logic to control max size of GRO packets.

```
    if (unlikely(p->len + len >= 65536 || NAPI_GRO_CB(skb)->flush))
        return -E2BIG;
```

It would need to be adjusted for IPv6 traffic using a new dev->gro_ipv6_max_size tunable (if skb has at least 8 bytes of headroom so that the extension header can be inserted between IPv6 network and TCP header) later in *ipv6_gro_complete*()

Changes in skb layout (optional)

Currently, MAX_SKB_FRAGS value is 17, meaning that we might have to use shinfo->frag_list more often.

Before Google adoption of 4K MTU, we internally increased MAX_SKB_FRAGS to 45 without ill effect.

Note that some NIC/drivers have hidden assumptions about MAX_SKB_FRAGS being “small” like 17.
bnx2x has in fact a limit of 13 page frags per TSO (MAX_FETCH_BD) but already has code to deal with this limitation.

skb layout (cont)

tcp sendmsg() tries to use 32KB page frags, so theoretically could reach ~512 KB per TSO packet.

Unfortunately zero copy paths use order-0 page frags, so would not be able to use Big TSO packets unless

- we increase MAX_SKB_FRAGS, or
- Generalize NETIF_F_FRAGLIST support and change TCP TX to use it (unlikely because this looks invasive change)

Driver changes in TX path

Some NIC have limits on number of TX descriptors per entry in TX ring, or might have various bugs. They can either implement a ndo_features_check() to drop NETIF_F_GSO_MASK from features, or try to split ‘too big’ TSO packets into smaller ones, maybe using a set of core helper functions.

(Reverting to standard GSO might be okay if the exceptions are not raised too often)

Cons ?

One of the issue one can think of is the increased RTT between peers, since the ‘airtime’ of packet trains can be multiplied by 2 or 4.

Possibility of hitting various bugs, especially if MAX_SKB_FRAGS is increased.

BQL becomes less effective. This might already be the case today with many TX queues.

Too many MSS per skb in TCP receive queue might slow down applications, because skb frags lookup is linear in recvmsg()