PATH to TCP 4K MTU and RX zerocopy

Netdev 0x14, August 2020

History of TCP zero copy in linux : TX (1)

Linux had sendfile() support since early days.

This is using ops->sendpage() or ops->sendpage_locked() and available to splice() users, like sendfile() and vmsplice()

In linux-4.14 Willem de Bruijn added MSG_ZEROCOPY support to sendmsg() system call, along with completions sent to the socket error queue.

sendmsg(MSG_ZEROCOPY) is slightly more efficient since it does not have to lock the socket for every page, unlike tcp_sendpage().

History of TCP zero copy in linux : TX (2)

Hardware features needed to support TX zero copy are limited to standard SG support and TX checksum offloading. There is no requirements on how memory blocks need to be sized/aligned.

Almost all modern NIC support these (and more)

TCP RX zero copy : context

Single TCP flow performance is limited by receiver, since sender can usually use both TSO super-packets and TX zero copy.

Optimal throughput over 100 Gbit link is 80 Gbit (Intel Cascade Lake)

```
 received 32768 MB in 3.54587 s, 77.5205 Gbit
 cpu usage user:0.025169 sys:3.08648, 94.96 usec per MB
```

About 70 % of the cpu cycles are spent in the kernel->user copy done in recvmsg()

What if... I could use mmap() instead of recvmsg()

Sure, this would be very nice, but this adds constraints.

Ideally, only the payload should be visible by the application, so all network headers shall not be mapped.

Then, MMU granularity is a page, and a page size is architecture dependent.

Seems fine, vast majority of servers use PAGE_SIZE=4K, at least for Google.

In absence of LRO, use 4K+ MTU

We can only map to user space completely filled pages, otherwise we could leak old data (security would be at risk)

Note that we can not attempt to clear holes, since skb frags are read-only (potentially shared) by design.

Solution : Use 4096 bytes of TCP payload per individual segment.

OK, 4096 bytes of payload, but… what about headers ?

As we said, the plan was to not give any headers to user space [1], so their size do not matter [2]

[1] A single mmap() should be needed to access a single contiguous piece of the TCP stream. Handling/skipping headers from user application would be not practical.

[2] Unless size matters. See next slide...

Interoperability

Header-data split: Ability for a NIC to dissect packets and place header and data into separate places.

Not all NIC implement header-data split, unfortunately.

Google has for instance a fair amount of servers using Mellanox CX-3 (mlx4)

Pseudo header-data split (1)

mlx4 does not have generic header-data split, but can use up to 4 segments to receive a packet. Each segment can be precisely sized.

Noting that more than 99% received packets in our DC are :
- TCP over IPv6, and 12 bytes of TCP options (TCP RFC 7323 TS option)

We can size the first fragment to be exactly **86 bytes** :

{Ethernet 14} {IPV6 40} {TCP with TS option 32}

Pseudo header-data split (2)

We copy this 86 byte first segment into skb->head, and attach one full page to the skb before it reaches GRO stack.

This has the nice side effect of avoiding further copies/pull in GRO stack.

- If received headers were **smaller** than 86 (eg IPv4 + UDP packet), then we have some payload in skb->head and nothing bad happens.

- If headers were **bigger**, stack will pull additional bytes on demand, thanks to pskb_may_pull().

Real header-data split... sort of...

Another NIC we use at Google has header-data split.

But no SG.

The headers are placed inline in the RX descriptor,
and the payload is placed into a single area,
which must be physically contiguous unless IOMMU is used.

Since we only want order-0 pages, this limits MTU to 4096+headers on x86.

So it looks like we use 86 bytes of headers

The ideal ready-to-be-mapped packet has therefore :

86 bytes of headers
4096 of payload -> MTU = (4096+86-14) = 4168

Since TCP could expand TCP option space in some cases (SACK blocks),
we had to limit the TCP ADVMSS to 4108 (4096 + 12)

This means that _if_ a data packet carries more than 12 byte of TCP option,
it won't be TCP rx zero copy candidate (payload wont fit a whole page)

Deployment strategy

Since we had to increase the device MTU, and there was no way all hosts could be atomically updated, we had to enable 4K traffic only for TCP for selected destinations. Default route is limited to small/safe MTU

```
ip link set dev eth0  mtu  4300
ip -6 route show
…
default … dev eth0 … mtu lock 1500
ip -6 rule | grep 550
550:from all iif lo ipproto tcp lookup 550
```

TCP negotiation

TCP 4K packets are enabled if both parties (active/passive) agree on using ADVMSS 4108.
We also raise WSCALE to 12 to get aligned RWIN values and avoid partially filled pages.

```
C->S: Flags [S], seq 69925, win 65535, options [mss 4108,sackOK,TS val 100 ecr 0,wscale 12], length 0

S->C: Flags [S.], seq 326, ack 69926, win 65535, options [mss 4108,sackOK,TS val 3220 ecr 100,wscale 12], length 0
```

Even if fabrics were changed to carry all these 4K+ packets, we had to implement a fast MTU fallback without relying on PMTU, with associated monitoring.

TCP tweaks

Switching to 4K MTU surfaced old issues caused by Delay ACK.

Take RPC using 2000 bytes :
With 1500 MTU, sending 2 segments elicits an immediate ACK.
With 4096+ MTU, sending one segment usually starts a delayed ACK timer.

-> Some benchmarks can exhibit radical differences because of this heuristic.

TCP TSO optimal size

tcp sendmsg() has logic to cook GSO packets using MSS multiples, constrained by the 64K limit of IP datagrams.

With MTU=1500, this leads to skbs with 45 segments of 1428 bytes of payload

With 4K+ MTU, we only can fit 15 segments per skb

-> 61440 bytes per skb, vs 64260 bytes per skb

Cons : skb->truesize increase

Going from MTU ~1500 to MTU ~4K has the effect of forcing the NIC driver to use 4KB allocations instead of 2KB to store incoming packets.

So ‘medium packets’ flows have {skb->truesize/skb->len} increase, and this can reduce effective RWIN (and thus throughput on long distance flows) by 50%.

However, once majority of flows have switched to 4K, truesize/ratio is almost perfect (no holes)

API in a nutshell

TCP stack support added in linux-4.18

See tcp_mmap.c in tools/testing/selftests/net/ for reference.

1) mmap() is used to reserve VMA space.
   tcp_mmap() makes sure pages will be Read Only.

2) getsockopt(fd, IPPROTO_TCP, TCP_ZEROCOPY_RECEIVE, &zc, &zc_len);
   To implement actual mapping of pages into user space.

Raw results

tcp_mmap easily reaches line rate (~100 Gbit), and has plenty of idle cycles. (100Gbit is about 3 Mpps at 4K MTU_

```
received 32768 MB (100 % mmap'ed) in 2.85541 s, 96.2656 Gbit
  cpu usage user:0.037939 sys:1.41831, 44.4413 usec per MB.
```

This means we probably could reach 200Gbit on capable links.

Note: Simply using 4K MTU brings nice performance improvements.

Cons ? (too many...)

MMU games are challenging in multi thread apps :

- mm->mmap_lock contention.
- Cost of TLB invalidation at madvise(MADV_DONTNEED) time.
- No Transparent hugepages (meaning one TLB miss per 4k page)
- XDP (does not like header-data split)
- Holding pages in user space VMA can defeat page pool recycling on the NIC, ultimately increasing pressure on MM zone spinlocks.

If application needs to access all the data, avoiding the copy in the kernel might not show significant benefit, depending on CPU caches size.

Various (upcoming) optimizations

We made several changes to speedup common operations.

- SO_RCVLOWAT changes to reduce number of EPOLLIN.
- Return INQ along with tcp receive zerocopy. And offer to copy sub-4K fragments without going through another recvmsg() syscall.
- vm_insert_pages() to insert multiple pages at once, reducing PTE spinlock acquisitions.
- Support the unmap operation outside of the getsockopt() call, while socket lock is not held.

Credits (of upstream contributions)

Eric Dumazet

Andy Lutomirski

Arjun Roy

Soheil Hassas Yeganeh