


# State of the union in TCP land

Eric Dumazet @ Google  
Netdev Ox16, Oct 2022



# One year of development

- Overall health status
- BIG TCP update
- HW-GRO + SW-GRO, and why RSC is broken
- Copy or not Copy ? (or the return of tx-nocache-copy !)
- TCP memory control / charging update
- network namespace optimizations
- TCP DDP / TCP Direct
- PSP, PLB
- TCP A-O, MD5 replacement

# linux TCP is still very active

- MPTCP got ~170 patches in last year
- TCP got ~270 patches in last year

Huge thanks to awesome contributors, reviewers and maintainers :

Kuniyuki Iwashima, Joanne Koong, Martin KaFai Lau, Jakub Kicinski, David S. Miller, Paolo Abeni, Willy Tarreau, Neal Cardwell, Coco Li, Cong Wang, Yuchung Cheng, Wei Wang, Talal Ahmad, Soheil Hassas Yeganeh, Arjun Roy, Menglong Dong, Akhmat Karakotov, Francesco Ruggeri, Alexander Duyck, Shakeel Butt, (and others ! ... *list is too long* ...)

# Few security/privacy issues addressed

Source port selection for active flows was using a too small secret table, as reported by security researchers Moshe Kol, Yossi Gilad, Amit Klein.

(Link: <https://arxiv.org/abs/2209.12993> )

Willy Tarreau made the secret much bigger and added more perturbations.

This made device tracking much more difficult, but I am sure more flaws will be discovered later :(

# BIG TCP update

BIG TCP was presented last year at Netdev 0x15  
( <https://netdevconf.info/0x15/slides/35/BIG%20TCP.pdf> )

Patches landed in linux-5.19.

Missing part in upstream kernel is the change to increase MAX\_SKB\_FRAGS to 45

Changing MAX\_SKB\_FRAGS could be challenging for some drivers/layers.

# BIG TCP at Google

With our nominal 4K MTU (or more exactly, TCP packets with 4096 bytes of payload, aka `advms` 4108)

A standard TSO/GRO packet contains 15 MSS (61440 bytes of payload).

We set `gro_max_size` and `gso_max_size` to ~181000, meaning TSO/GRO can use 44 segments (180224 of payload). This value was chosen so that GRO would not have to use `shinfo->frag_list`, so that TCP receive zerocopy is still functional.

# BIG TCP : performance on 200Gbit links

Using Intel Xeon Platinum cpus

Nominal one TCP flow throughput (with standard netperf benchmark)

Without BIG TCP : around **90 Gbit**

With BIG TCP : around **120 Gbit**

Standard tcp\_mmap TCP receive zero copy test reaches **170 Gbit**.

Major bottleneck is memory copies, can we do better ?

## HW-GRO + SW-GRO

HW-GRO packets provided by NIC are likely limited to 64KB of payload.

In order to support BIG-TCP and hw-gro, we had to be able to coalesce hw-gro provided GRO packets together in SW-GRO.

Even without BIG-TCP, this feature is useful because can terminate their GRO packets too soon, depending on tight memory buffers and timeouts.

This changed landed in linux-6.1 “gro: add support of (hw)gro packets to gro stack”



# HW-GRO / RSC is silly

HW-GRO is often based on RSC specs from Microsoft.

<https://learn.microsoft.com/en-us/windows-hardware/drivers/network/rules-for-coalescing-tcp-ip-packets>

Unfortunately, there is very little support for TCP options.

Anything else than “standard” TCP TS (RFC 7323) is not aggregated.

This makes little sense, because on TX side, TSO is happily copying all TCP options regardless of their content into all generated segments/MSS

# How to fix RSC

Allow aggregation of frames if they share the exact same TCP options values.

Trying to allow complex games about TCP TS Tsval/Tsecr is simply useless at current network speed.

Senders are likely using TSO, RSC should be able to build equivalent GRO packets.

# Copy or not Copy ?

We have TCP zero copy for both TX and RX, but... It is complicated.

TX path has issues when applications use hugepages, because of false sharing on the compound head page refcount. (MM bottleneck)

RX path also has MM bottlenecks on heavily multi-threaded applications.

x86 4K page size is too small :/

Recent cpus are getting higher memory throughput as shown in previous slide.

# AMD (interesting ?) case

AMD EPYC cpus (Rome ...) lack DDIO (due to the cpu architecture)

On TX path, NIC performs DMA from main memory to the NIC.

On RX path, NIC performs DMA to main memory.

tcp sendmsg() is copying data from user space, to kernel pages, polluting cpu caches to store the destination. DMA from the NIC has to stall while cpu is flushing its cache. Meanwhile huge parts of CPU caches were evicted.

“ethtool -K eth0 tx-nocache-copy on” to the rescue ?

# tx-no-cache on and off, and on...

Tom Herbert added tx-nocache-copy option in linux-3.0 (2011)

Later in linux-3.14 (2014), Benjamin Poirier changed the default value to off, because typical servers at that time were using Intel and DDIO. Populating the L3 cache was allowing the NIC to perform a much faster DMA access, and the NIC would anyway force the data to reside in L3 cache even if CPU was avoiding it at user->kernel copy time.

Guess what ? With AMD EPYC cpus, flipping tx-nocache-copy on again makes quite a difference :)

# Can we turn on tx-nocache-copy on AMD cpus

Probably.

This could use a sysctl, turned on by default at boot time when AMD cpus are detected.

Then register\_netdevice() could use this sysctl to automatically enable NETIF\_F\_NOCACHE\_COPY feature for non virtual interfaces.

# tx-nocache-copy results on AMD

Experiments on a 100Gbit NIC testbed, AMD Rome, using one TCP flow (one user thread doing the tcp sendmsg())

```
ethtool -K eth1 tx-nocache-copy off
```

-> Max throughput : **~78Gbit**

```
ethtool -K eth1 tx-nocache-copy on
```

-> Max throughput: **line rate (100Gbit)**

# What about RX side (on AMD again)

NIC DMA incoming packets into main memory.

Then `tcp recvmsg()` copies from the kernel buffer the payload into user buffers, while using standard loads/stores, wasting precious cache lines in cpu caches, while the source part is guaranteed to not be reused (kernel buffer is freed/recycled for next packet to come)

`tx-nocache-copy` on x86 is using `MOVNTI` instruction, which gives a hint to the cpu to minimize cache pollution during the write to memory.

Could we use the same strategy for RX path ?



# MOVNTI (store register to memory)

Unfortunately, there is no standard x86 instruction to perform a

Load memory into register, with a non-temporal hint to minimize cache pollution.

Only AVX provides such a functionality, with **MOVNTDQA**

# MOVNTDQA

Unfortunately, using AVX in the kernel might be complicated.

MOVNTDQA has a 16byte alignment constraint, meaning that a preamble code would need to eventually copy up to 15 bytes with standard load/store before using AVX (fortunately the store part has no alignment requirements)

Using MMX registers might need to save/restore FPU registers, this could very well be too expensive (even if we only need one or two XMM registers !)

Ongoing investigations/tests at Google (Wei Wang and Jeffrey Ji)

# TCP memory control

TCP sockets can consume memory for TX/RTX queues, and receive queues.

linux was using a 'forward allocation' cache **per TCP socket**, avoiding too many updates on a global variable (`tcp_memory_allocated`) tracking total TCP memory usage.

This caused too many OOM issues on servers with many TCP sockets.

In linux-6.0 we got rid of excessive per-socket forward allocations, keeping them to the minimal value, and went instead to a **per-cpu** cache.

# TCP memory control & memcg

Removal of per-socket forward allocation caches caused a regression for memcg enabled workloads. Shakeel Butt made the change to memcg to better deal with more memcg alloc/free interactions from TCP stack.

Our goal is to no longer rely on global `tcp_mem[]` control, and instead use memcg to control/limit TCP buffers.

This means a network intensive application could use its memory budget to safely increase TCP socket memory footprint if needed.

# Network namespace optimizations

Kuniyuki Iwashima added in linux-6.1 optional per net-ns TCP hash table, for better isolation among netns. It could also improve performance for specific workloads, allowing either a very small hash table, or a very large one.

Joanne Koong added a bhash2 table hashed by port and address, to speed up bind() for many sockets bound to the same port.

# Ongoing developments

TCP DDP, TCP Direct: Direct placement of payload in GPU memory, while headers are still stored in host memory. Standard linux kernel TCP stack is used. This also can allow a more generic 'TCP receive zerocopy' with no MMU (page flipping) games.

PSP

<https://cloud.google.com/blog/products/identity-security/announcing-psp-security-protocol-is-now-open-source>).

Hopefully TCP implementation can be upstreamed soon.

# PLB, TCP-AO, data-locality optim

PLB (Protective Load Balancing,

<https://dl.acm.org/doi/10.1145/3544216.3544226> )

should land in linux-6.2 (patch series from Mubashir Adnan Qureshi)

TCP-AO (replacing/extending TCP MD5) seems to be in active development, with two different (big :/) patch sets.

I am still working on better tcp socket data layout to minimize number of cache lines read/written per TCP action.

# Conclusions

TCP is still strong, and gets constant improvements.

We often hit various bottlenecks in MM layer, scheduler layer, platform memory-bw, interrupt latency, so we have many challenges to address in the future.

Please join us to make linux TCP even better.

Thanks !