TCP/misc works

Eric Dumazet @ Google

1) TCP zero copy receive
2) SO_SNDBUF model in linux TCP (aka better TCP_NOTSENT_LOWAT)
3) ACK compression
4) PSH flag set on every TSO packet

Design for TCP RX ZeroCopy : 4KB frames + mmap()

- First part : 4096 bytes TCP payload per Ethernet frame (instead of ~1448)
  - Header split: (Ethernet, IPv{4|6}, TCP) are placed into a separate zone (skb->head)
  - Efficient memory usage (no more ‘holes’), 22->32Gbit throughput on a single TCP flow on mlx4.
  - Most modern NIC are programmable to perform Header Split.
    Mlx4 (prototype patch using IPv6), Intel NICS, GoogleQ100.
  - No changes in applications / upper stacks.

- **Blocker**: Fabric ~4K MTU support
  - Will double throughput and halve latency on its own : A GRO packet needs only 15 frags, instead of 45. Less atomic operations at skb free time.

mmap() support for TCP

- Addr = mmap(NULL, size, PROT_READ, MAP_SHARED, fd, 0);
- PAGE_SIZE=4096 arches like x86.
  - Make TCP input data available to user without any copies, in read-only mappings.
  - ~100 lines of code change.
  - Could be used in Cloud VM as well.

- Easier than TCP TX ZeroCopy to use, with significantly lower overheads.
  - No release signals required (unlike TX ZeroCopy, does not need wait for ack from the other side).
  - No error queue overheads.

- Might be a challenge for single-process many-threads applications
  - Per process VM semaphore contention, TLB flushes at munmap() time.

Early results

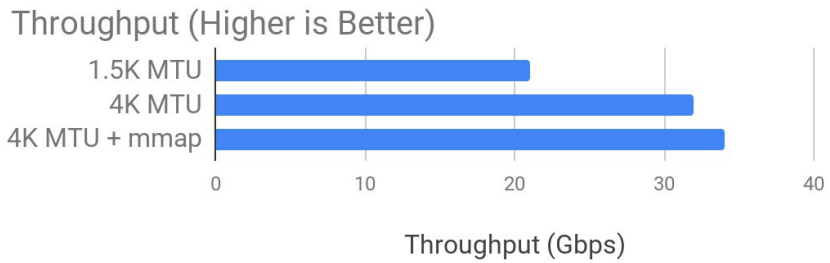
netperf (TCP_STREAM) one flow (throughput and cpu time per MB)

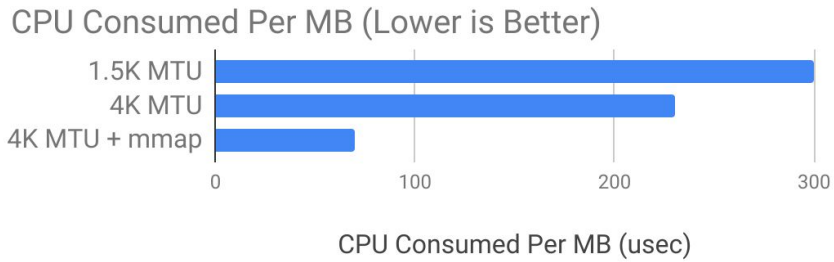
MTU=1500 ~21 Gbit (300 usec/MB)

MTU = 4KB+ ~32 Gbit (230 usec/MB)

MTU = 4KB+ + mmap() ~34Gbit (60 usec/MB)

Some graphs :


Throughput (Higher is Better)
1.5K MTU
4K MTU
4K MTU + mmap
0
10
20
30
40
Throughput (Gbps)


CPU Consumed Per MB (Lower is Better)
1.5K MTU
4K MTU
4K MTU + mmap
0
100
200
300
CPU Consumed Per MB (usec)

Missing features in TCP vs RX zero copy

Ability to deploy the feature gradually. At Google we chose to let our network manager control the feature using eBPF hook at connect()/accept() time, instead of a route attribute.

TCP_MAXSEG includes the TCP option space, so can not really be used to force the payload to be 4096 bytes exactly : Application can not control if TCP TS option is going to be used, or if (a variable number of) SACK will be added by TCP stack.
In order to deal with blackholes, need the ability to perform reverse of PMTU probes.

2) SO_SNDBUF model in linux TCP

Per socket sk->sk_sndbuf can be either set by the application to a static value via

setsockopt(fd, SOL_SOCKET, SO_SNDBUF, &sndbuf, sizeof(sndbuf));

Or automatically tuned by the kernel, depending on the observed usage of the rtx+write queue, up to sysctl **tcp_wmem[2]**.

Most applications just leave autotuning in place.

Current default value for tcp_wmem[2] is 4 MB

We have plans to increase it to higher values, like 16 MB or even 64 MB to increase max throughput on long distance flows.

Problem is this could allow applications to fill more data in the write queue of thousands of sockets.
This would cause extra memory consumption, leading to unexpected OOM

Years ago I added TCP_NOTSENT_LOWAT support in linux-3.12

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c9bee3b7fdecb0c1d070c7b54113b3bdfb9a3d36

Current implementation suffers from following flaws :

- Too many context switches
- Missing wakeups when SACK allow to drain the write queue below the NOTSENT_LOWAT

Once these problems fixed, we could increase tcp_wmem[2] _and_ defaults notsent_lowat to 1MB or so.

Too many context switches

1) exact tp->notsent_lowat value is used to allow tcp_sendmsg() to cook another skb, and same value is used to generate EPOLLOUT events.
   This means we might generate one EPOLLOUT for every skb that is cleaned up (from rtx queue)

This can be solved by using tp->notsent_lowat*3/2 at sendmsg() time, and tp->notsent_lowat/2 at tcp_poll() time.
We would then generate one EPOLLOUT per tp->notsent_lowat

EPOLLOUT is generated only when packet is cleaned up from rtx queue, or when sk->sk_sndbuf is expanded (a few times in socket lifetime really when autotuning is making forward progress)

Details : sk_wmem_free_skb(sk, skb) is called when skb if removed from rtx queue,
it sets SOCK_QUEUE_SHRUNK socket flag.
Then later tcp_check_space() checks SOCK_QUEUE_SHRUNK and call tcp_new_space() which then calls sk->sk_write_space(sk);
This means that if TCP receives SACK and is able to send more fresh packets, moving them from write queue to rtx queue, we do not actually call sk->sk_write_space(sk). This can lead to have no more notsent bytes/skb in write queue and a stall of the pipe.

One solution is to add to the bottom of tcp_write_xmit() the following :

```
    if (sent_pkts) {
        ....
        tcp_schedule_loss_probe();
+       if (ca_state >= CA_recovery)
+           set SOCK_QUEUE_SHRUNK;
     }
```

3) Too many TCP ACKS

Too many acks are killing wifi performance, but also adds immense pressure in servers in data centers. Cost of sending and receiving one ack is about 2 usec of cpu cycles and a fair share of memory traffic. Also some expensive network equipments have a per packet cost.

Most TCP ACKS are redundant. Lack of GRO (most wifi NIC do not have GRO, or bandwidth is too small for GRO to kick in) is the reason we have so many ACKS.

Linux 4.18 added SACK compression, thanks to a new per-socket high resolution timer. (“tcp: add SACK compression”)

ACK compression : Future work

My plan is to extend the use of this high resolution timer for other cases where delaying ACKS might save network capacity and cpu cycles.

And eventually get rid of the legacy ‘delayed ack jiffies-based timer’, but this might be not worth the trouble.

4) PSH flag set on every TSO packet

This hack has been used for several years at Google, to help GRO engine on receivers not holding a packet.
This is for two reasons :
- We use usec resolution TCP TS options (tsval/tsecr)
- We relieve GRO to not hold a packet if the sender gives a hint that following MSS might come way too late for GRO to have a chance to add another MSS to current GRO packet.

-> This helps reducing number of GRO packets to parse during GRO processing, thus reducing GRO overhead.