


# State of the union in TCP land

Eric Dumazet @ Google  
Netdev 0x19, March 2025



# Agenda

- Overall health status
- TCP Packetdrill tests
- Better cache locality
- connect() improvements
- TCP devmem
- **Poor man's RFS**
- **Atomix are tooooo expensive**
- BQL under high stress and many TX queues ?
- TCP Swift (oops)

# linux TCP activity

- MPTCP got ~200 patches in last 8 months.
- TCP got ~140 patches in last 8 months.

Many thanks to all contributors, reviewers and maintainers.

- **63 different authors during this time period.**
- Neal Cardwell became an official TCP maintainer.
- Kuniyuki Iwashima became an official reviewer.

# TCP selftests

Willem de Bruijn added ~70 packetdrill tests to kselftests.

More to come hopefully.

# Better memory locality for sockets

Long effort to reorganize fields to lower the number of cache lines in TCP fast path.

This is essentially depending on DCCP removal, so that we can move fields from *struct inet\_connection\_sock* to more appropriate points in tcp\_sock.

Although following fields could be moved right away :

**icsk\_ulp\_ops, icsk\_ulp\_data, icsk\_clean\_acked, icsk\_sync\_mss**

# connect() optimization

linux-6.15 will have big improvements for connect().

The search for an available sport was using expensive spinlocks per each attempt. It is now using RCU for better scalability.

- > Up to 600% increase of performance in tcp\_crr synthetic tests.
- Same can be done for for bind(), although it is probably not an urgent thing after IP\_BIND\_ADDRESS\_NO\_PORT adoption.

# TCP devmem

RX path upstreamed in linux-6.12

Mina is still working on submitting the TX path.

# RSS, RFS, aRFS

RSS : Receive Side Scaling : spreads packets on multiple receive queues based on a 'hash' function using the TCP 4-tuple (usually Toeplitz hash)

RFS : software layer trying to steer the packets after they were received on the host, after RSS made its decision. Adds more flexibility, but still at a high cost, especially on NUMA platforms.

aRFS : some kind of offload mechanism so that the NIC can select a more optimal queue, based on which queue tx packets went through.



# Poor man's aRFS

aRFS is hard to implement : It needs hardware support and to store state on the NIC, on a per-flow basis. Changing the queue # when a flow migrates (after a thread cpu migration) is expensive.

For IPv6 flow, the 4-tuple uses  $128+16+128+16$  bits.

Poor man's RFS is to mask  $n$  bits from the addresses when TCP peer receives a request from the remote peer about a wrong placement.

NIC does not have to implement aRFS, the peers are using more than one IPv6 address.

# unload RFS

To make that happen, a TCP option can be added in ACK whenever a peer receives a packet on the wrong NUMA node, or wrong RX queue.

When receiving this option, the peer picks another source IP address for future packets. This could either be random if NIC use Toeplitz hash, or a more direct approach for XOR-like hash functions.

This would slightly confuse some diagnostic tools like tcpdump and tcptrace, but possible gains would be quite big.

# atomic operations are a major bottleneck

MOTD : Intel(R) Xeon(R) 6985P-C Granite Rapids

120 cores per socket 540MB of cache (3 dies with 40 cores)

**Total of M=480 cpus** on a dual-socket host.

**N=6 NUMA nodes.**

30 RX/TX queues on the 200Gb NIC.

Mechanisms involving passing objects from cpu A to B no longer scale unless appropriate batching is used.

# Be careful about atomics

perf can help us to find most of them:

```
perf record --pfm-events gnr::MEM_INST_RETIRED:LOCK_LOADS
```

**Extreme case** : If 480 cpus attempt an atomic\_add on a shared location, they can perform about 20,000,000 operations per second, and some of unlucky cpus can observe a 3ms latency on each instruction.

# Do not assume atomic operations are fair

```
perf stat atomic_bench -T 480 # 2 second run
```

```
The atomic counter is 24421936, total_cycles=1402450238634, 57425 avg cycles per update
```

```
[05] ***** 5953613 in [32,64[ cycles (51 avg)
[06]                                16730 in [64,128[ cycles (71 avg)
[07] ***                                389453 in [128,256[ cycles (180 avg)
[08]                                37445 in [256,512[ cycles (375 avg)
[09]                                56607 in [512,1024[ cycles (762 avg)
[10] *                                180324 in [1024,2048[ cycles (1530 avg)
[11] ****                             517527 in [2048,4096[ cycles (3167 avg)
[12] *****                         1930952 in [4096,8192[ cycles (6212 avg)
[13] *****                         3495471 in [8192,16384[ cycles (12097 avg)
[14] *****                         3545172 in [16384,32768[ cycles (24540 avg)
[15] *****                         4187656 in [32768,65536[ cycles (46021 avg)
[16] *****                         1995526 in [65536,131072[ cycles (91136 avg)
[17] *****                         976710 in [131072,262144[ cycles (177971 avg)
[18] **                             276319 in [262144,524288[ cycles (348487 avg)
[19] *****                         729759 in [524288,1048576[ cycles (702747 avg)
[20]                                24834 in [1048576,2097152[ cycles (1384628 avg)
[21]                                18932 in [2097152,4194304[ cycles (3533098 avg)
```

# perf output:

Performance counter stats for './atomic\_bench -T 480':

927,884.81 msec	task-clock	#	444.296 CPUs utilized
8,237	context-switches	#	8.877 M/sec
1,392	cpu-migrations	#	1.500 M/sec
1,740	page-faults	#	1.875 M/sec
3,063,891,355,597	cycles	#	3302019.817 GHz
16,854,153,430	instructions	#	0.01 insn per cycle
2,384,773,031	branches	#	2570119.790 M/sec
4,628,335	branch-misses	#	0.19%

# syslog output

```
[90058.275011] INFO: NMI handler (perf_event_nmi_handler) took too long to run: 1.919 msecs  
[90058.275561] INFO: NMI handler (perf_event_nmi_handler) took too long to run: 1.993 msecs  
[90058.275823] INFO: NMI handler (perf_event_nmi_handler) took too long to run: 2.016 msecs  
[90058.277814] INFO: NMI handler (perf_event_nmi_handler) took too long to run: 2.121 msecs  
[90058.280299] INFO: NMI handler (perf_event_nmi_handler) took too long to run: 2.343 msecs  
[90058.284205] INFO: NMI handler (perf_event_nmi_handler) took too long to run: 2.389 msecs
```

# Some atomic can hide

```
tcp_check_space()
{
/* pairs with tcp_poll() */
  smp_mb(); // ouch
...
}
```

We probably can do better.

( smp\_mb() -> smp\_mb\_\_after\_atomic)



# time to add hierarchical per-cpu-counter ?

Instead of : 1 shared-big-counter + M per-cpu-small-counter

Add an intermediate per-NUMA-counter :

1 shared-big-counter + N medium-counter + M per-cpu-small-counter

-> reduce number of updates over the shared-big-counter

Other possibility : per-NUMA-counter to reduce memory needs for SNMP.

# One bad case : `skb_attempt_defer_free`

It queues one packet to a remote cpu `softnet_data->defer_list`, using a spinlock.

Under arbitrary load, up to 479 cpus might attempt to queue one skb at a time.

Under a typical 20 Mpps workload, the added spinlock and false sharing on the ~30 spinlocks is clearly visible.

To fix this, we need more suitable cache to batch ~8 skbs per spinlock acquisition. SLAB had the concept of alien caches, adding NUMA awareness to avoid quadratic costs ( $M \times M \times \text{sizeof}(\text{cache})$ ) :  $M \times N \times \text{sizeof}(\text{cache})$

# BQL with many TX queues and high stress

BQL is a a nice thing up to 10Gb

Being tied to how often the `napi_poll()` is running, it becomes less efficient under load when `napi_poll()` process large batches, and/or if `napi_poll()` is lagging.

Is it worth the costs for servers using FQ/pacing ?

# TCP-Swift

Kevin Yang ([yyd@google.com](mailto:yyd@google.com)) plans to upstream TCP-Swift congestion control in 2025.

Swift is a delay-based congestion control algorithm that uses AIMD based on network\_rtt measurements.

It is designed for datacenter networks where a round-trip propagation delay is usually around 100us or less.

<https://dl.acm.org/doi/10.1145/3387514.3406591>