Optimizing UDP for content delivery: GSO, pacing and zerocopy

Willem de Bruijn
willemb@google.com

Eric Dumazet
edumazet@google.com

Abstract

UDP is a popular basis for the experimentation with and rapid deployment of new protocols. Experience shows a large gap in cycle efficiency compared to in-kernel TCP.

This talk presents recent optimizations to the UDP stack that narrow this gap. At the core are optimizations long available to TCP: segmentation offload, pacing and zerocopy. We present the recently merged UDP GSO and SO_TXTIME interfaces and discuss select implementation details. We also review partial GSO as it fits in this context and discuss optimizations that are in submission: UDP zerocopy transmit with MSG_ZEROCOPY and UDP GRO.

Introduction

UDP is common in video streaming and other time critical protocols for which timely unreliable delivery is a feature. It is also uniquely positioned for broader experimentation with and rapid deployment of novel higher layer protocols. It is widely available across operating systems, requires no superuser privileges and is well supported in the network by middleboxes. Exposing only a thin unreliable datagram service over IP allows varied application specific extensions.

A recent example of such a protocol is QUIC [1], a transport layer protocol that combines reliable delivery and congestion control with stream multiplexing, header compression, cryptography and low latency connection establishment. QUIC is widely deployed at Google, serving over 35% of all Google egress traffic [2]. Key to rapid deployment and iteration has been being able to ship QUIC directly with applications, e.g., in the Chrome browser.

We have previously measured transmission over QUIC to cost up to 3.5x the CPU cycles per byte versus transmission over TCP [2]. QUIC is going through IETF standardization. An obvious next step would be to write reimplement the protocol as a first class operating system primitive, in the kernel. This would it offer tangible benefits in terms of cycle efficiency.

A kernel implementation only improves this single protocol, ignoring many of the benefits of UDP. Instead, in this paper we investigate improvements to UDP as a service that benefits the wider ecosystem of established and future UDP based protocols. We do focus on cycle efficiency and content delivery, with QUIC as the demonstration protocol and Youtube streaming as the target use case. In this scenario, a representative workload is a server that communicates with tens of thousands of clients at time, sending a relatively low rate stream of around 1 MBps to each.

The paper discusses the various relevant recent improvements to the UDP send and receive paths. Specifically, generic segmentation offload (UDP GSO) and partial GSO, UDP zerocopy, earliest delivery time (EDT) scheduling with SO_TXTIME and FQ and generic receive offload (UDP GRO). We complete in Section by discussing how this affects a QUIC server software architecture, comparing the previously outlined Google design [2] with one based solely on UDP. The paper includes considerable work by people besides the authors. We thank them in the acknowledgments section at the end.

Efficiency

Some of the cycle cost in serving QUIC was due to inefficiencies in the application that are out of scope here. After those are resolved, the remainder was about 2x cycle increase. This is for the entire application, which indicates an even higher cost for the transmit path in isolation. We investigated the cost of serving over UDP vs TCP independent of QUIC with a simple benchmark, `udpgso_bench`, which is available in the kernel source tree. In short, a standard TCP configuration sees a nearly 5x speed-up over UDP. Figure 1 shows the exact numbers. We will return to these in more detail. See also the appendix for test setup and details.

A cycle profile with `perf-tools` indicates no single obvious hotspot in the UDP path that explains the cost difference. Instead, cycles accrue across the stack due to a much higher number of stack traversals for the same number of bytes.

The main difference between the two protocols in this regard is in TCP segmentation offload (TSO). Segmentation offload allows the protocol stack to pretend that a network device can send a much larger packet than is actually supported by the network path. This results in fewer network stack traversal on transmission. The network device breaks apart the large TSO packets down to MTU, replicates and fixes up the network and transport headers. With TSO, the TCP stack send packets of the maximum size allowed by the underlying network protocol, 64 KB (including the network header for IPv4, excluding the header for IPv6), to the device.

If the network card does not support TSO, the Linux kernel stack can perform this operation just before passing packets

	Gbps	calls/s	Mcycles/s	Speed-up (%)
TCP	9.3	19040	2800	100
TCP gso	LR	19040	1856	162
TCP tso	LR	19040	618	487
UDP	LR	812000	2801	107
UDP gso	LR	18248	1727	174

Figure 1: Cycle cost of `udpgso_bench_tx`

to the device driver. This generic segmentation (GSO) layer emulates NIC TSO. Doing so costs CPU cycles, but the savings are still considerable over passing MTU sized packets through the entire stack (for a standard 1500B MTU). This can easily be quantified for TCP, as TSO and GSO can be individually enabled with `ethtool -K`. The first three rows in Figure 1 compare TCP without segmentation [1], with GSO and with TSO. The cycle improvements are stark. GSO is 1.6x more efficient, TSO 4.9x. The abbreviation LR denotes linerate for this 10Gbps machine. TCP without segmentation does not saturate the link for this single threaded benchmark.

The cycle savings accrue across the stack, not only from reduced system call invocation. To achieve the same for UDP, then, it is not sufficient to batch system calls with `sendmmsg`. The protocol needs true segmentation offload.

UDP GSO

UDP GSO, like TSO, makes it possible for applications to generate network packets for a virtual MTU much greater than the real one. Size is limited only by the underlying network and link layer protocol bounds. Unlike TCP, UDP is not a bytestream protocol, which does add an extra constraint. Even without TSO, a process can send Megabytes of data to a TCP socket in a single system call. This is not the case for UDP, as it implicitly signals datagram message boundaries through the `send` buffer size.

With UDP GSO, the length passed on `send` no longer matches the expected length on the wire. This datagram *gso size* now must be explicitly signaled with an extra argument to the send call. The kernel constructs the larger datagram, passes it to the UDP and IP layers wit this metadata attached. The network device or GSO layer takes the large datagram, splits up the payload in gso sized segments and replicates the header. A key point is that the UDP and IP headers are easy to replicate, bar for the UDP checksum field which the NIC already computes.

UDP GSO is not the same as UDP fragmentation offload. UFO also builds a larger than MTU datagram. But it does not segment them into individual UDP datagrams. It sends the single large UDP datagram on the wire as a sequence of IP fragments. Fragmentation is not a feasible strategy for large scale content delivery. Reassembly is expensive receive side and resources are purposely limited to withstand fragmentation DoS attacks.

[1] After reverting commit 0a6b2a1dc2a2 ("tcp: switch to GSO being always on")

Interface

The Linux kernel has a UDP GSO implementation as of version 4.18 [3]. The application explicitly notifies the kernel of the size of the datagram payload as it should appear on the wire either through a socket option or passed along as control message with the `sendmsg` call, as shown in Figure 2. For more examples, see the selftests in the kernel tree at `/tools/testing/selftests/net/udpgso.c`.

To qualify for GSO a few prerequisites need to be met.

Size The `gso_size` shown in Figure 2 is a common choice, based on the default Ethernet link layer data frame. The value must always be chosen to build packets that fit the link layer constraints indeed. It may have to be even smaller, if the route or path MTU is lower than that of the device.

Without GSO, a UDP `send` call fails by default if it exceeds path MTU – though this policy is configurable with socket option `IP_MTU_DISCOVER`. UDP GSO follows these same rules, but applies them to parameter `gso_size` instead of the size of the send buffer. Also, unlike for the TCP bytestream the total send buffer must always fit in a single network packet. So it is bounded by the maximum packet size of the underlying network layer, 64KB including network header for IPv4, 64KB excluding that header for IPv6.

The gso size parameter communicates the length of the payload without network or transport layer headers. A `gso_size` all the way down to 1 is a valid choice. A value of 0 effectively disables GSO. The sum of network header length, UDP header length and gso size must be smaller than or equal to MTU as defined by the constraints of `IP_MTU_DISCOVER`.

Segments The total number of segments is capped to UDP_MAX_SEGMENTS to avoid having untrusted processes be able to burst at unreasonable packet rates. This limit is set to 64, the first power of two larger than the IP maximum of 64KB divided by widely used Ethernet MTU of 1500.

Send buffer size must exceed `gso_size`. The GSO layer rejects packets that are smaller than *or equal to* the gso size. This is not a UDP specific constraint. It is not possible, therefore, to simply set the socket option on a socket that sends both gso and regular datagrams. In practice, passing `gso_size` as a control message is often preferable to the socket option, also when multiplexing multiple flows with possibly different path MTUs over a single unconnected socket. The payload length must thus be greater than `gso_size`. But it does not have to be an exact multiple. If it is not, the last segment will be shorter than the others and its IP and UDP length parameters adjusted as needed.

To see more edge cases that demonstrate these various constraints, see the previously mentioned selftest.

Checksum The GSO layer splits a large datagram into MTU sized chunks and adjusts its network and transport layer headers. For UDP, The source and destination port fields are the same for all packets. The length is, too, except for the just

Socket option

```
int gso_size = ETH_DATA_LEN - sizeof(struct ipv6hdr) - sizeof(struct udphdr);

if (setsockopt(fd, SOL_UDP, UDP_SEGMENT, &gso_size, sizeof(gso_size)))
        error(1, errno, "setsockopt udp segment");
```

Control message

```
cm = CMSG_FIRSTHDR(&msg);
cm->cmsg_level = SOL_UDP;
cm->cmsg_type = UDP_SEGMENT;
cm->cmsg_len = CMSG_LEN(sizeof(uint16_t));
*((uint16_t *) CMSG_DATA(cm)) = gso_size;

sendmsg(fd, &msg, 0);
```

Figure 2: UDP GSO interface: pass gso size through setsockopt or cmsg

discussed case. The the checksum field, however, decidedly is not.

UDP GSO requires a device with UDP checksum offload support. This is the vast majority of modern hardware. If checksum offload is not available, the operation could be transparently performed in the GSO layer. But, it is generally much cheaper to compute the checksum simultaneously while copying the data from userspace in `send`. With GSO, this checksum would have to be thrown away and a new one computed for each segment. Worse, at GSO time, the relevant data may very well no longer be in the cache. To avoid accidental performance degradation due to reloading of every byte of payload, UDP GSO fails on a path without checksum offload to nudge the user to revert to non-segmented send and use the copy-and-checksum optimization.

Hardware Support

GSO does not reach the full potential of segmentation offload, as the experiment in TCP GSO versus TSO demonstrated. There is nothing precluding hardware support in principle. UDP GSO specifies a single gso size for all segments. Subject to device descriptor specifics, this can relatively easily be passed along to the NIC. Alexander Duyck demonstrated hardware support within two weeks of UDP GSO being merged, with an RFC patchset for the Intel ixgbe driver [4]. The Mellanox mlx5 supports the feature in hardware as of kernel 4.19 [5, 6]. The authors report numbers that corroborate our results in Figure 1, a 2x improvement from GSO, but another 3x improvement over that with hardware UDP LSO (1.5x higher throughput at half the CPU cycles).

Partial GSO The main practical impediment to offloading to hardware is handling the last segment [7]. Alexander had previously already solved that in another context. If the payload is not a multiple of gso size, then both network header and udp header have to have their length fields adjusted for this last segment. Alexander implemented an elegant workaround for devices that cannot support this. `GSO_PARTIAL` is a feature in the software GSO layer, so available regardless of hardware device. It detects when a UDP GSO packet is not an exact multiple of gso size and en route to a device that is capable of UDP GSO but not of updating these fields. In that case, it splits the GSO packet in two packets, one GSO that is a multiple of its gso size and one regular packet that already has its headers adjusted. Partial GSO does not have to be configured. It is applied automatically depending on hardware features.

Zerocopy

Segmentation offload opens up another optimization that TCP has had for a while, but was impractical for UDP at MTU size. After applying GSO, the program has a much clearer hot spot to further optimize. Profile data shows that `copy_user_fast_string` is the hottest function in the trace at 13% of all cycles.

Zerocopy transmission with `MSG_ZEROCOPY` [8] avoids copying of payload. Instead, it pins pages and notifies the process when data is no longer in use by the kernel. These two operations per call are not free. For this reason the potential for savings from copy avoidance grows as the bytes per call increases. The first RFC patch sets included UDP protocol support. This was dropped from the final version because at 1500B MTU we saw no benefit. The UDP GSO RFC patchset included an updated version of zerocopy support [9].

One caveat to copy avoidance is that cycle savings from avoiding data loads and stores are highly dependent on whether the data is in the cache. For modern protocols that encrypt every byte, the data might be warm so that the cost of the memory loads are low. On the store side, the operations may be non-temporal and not affect the cache hit rate of other tasks. In practice we have seen benefits of zerocopy even for protocols that encrypt, as the application transmit path in these cases is often complex and the encryption operation does not happen necessarily at the time of the `send` call (e.g., due to framing, pacing, and socket backpressure).

Figure 3 extends the comparison of TCP and UDP with copy avoidance. It shows the relative time spent copying from userspace without zerocopy as obtained with `perf record`,

		Copy	Zerocopy	Speed-up
	copy%	Mcyc/s	Mcyc/s	%
TCP	4.35	2800	2800	100
TCP gso	10.3	1856	1704	109
TCP tso	26.7	618	425	145
UDP	3.11	2800	2800	100
UDP gso	13.4	1727	1690	102
UDP gso (CT)	21.2	1916	1694	113

Figure 3: Cycle cost with copy avoidance (Mcycles/s)

the total cycles spent without zerocopy and the total cycles with zerocopy. These are cpu-wide cycle measurements that include all kernel processing (`-a -C 5`). A profile of just the process cycles would make copy cost seem larger than it is end-to-end, especially with such a minimal benchmark application. Copy avoidance is indeed not effective for UDP without GSO. The payload is too small to warrant the extra pinning and notification cost. This does not change considerably with UDP GSO for this benchmark. But this is to be expected, as it reuses the same send buffer over and over again. Running the test with a cache thrashing (CT) variant that rotates buffers better demonstrates the gains when copying cold data. Copy time goes up from 13% to 21%. As expected, in copy mode, so does the total cycle cost. With zerocopy, it stays flat – and relative improvement increase from 2% to 13%. TCP can send even larger blocks than 64KB at a time to further amortize the notification cost. UDP cannot, so these numbers may or may not show the upper bound on benefits from zerocopy. It depends on how much higher copy overhead can be than 21%. The difference in time spent in copying between TCP GSO and TCP TSO indicates potential higher savings for UDP with full hardware segmentation offload, too.

Pacing

Segmentation builds a train of packets and sends it out at once. In the extreme case, 64 segments are sent together where they previously may have been spaced out simply by the latency incurred in the application and kernel transmit paths for each datagram.

Even for a relatively low rate flow, bursts can cause queue overflow in routers, as the rate is high at small timescales. In time sensitive protocols, this lowers service quality as packets are dropped. For reliable delivery like QUIC, it adds network load due to retransmissions. These also add CPU load, negating some of the gains of segmentation offload.

Effective delivery requires pacing of a flow to its intended delivery rate even at small time scale. The delivery rate is generally chosen by the congestion control algorithm. Linux exposes the `SO_MAX_PACING_RATE` socket option to further bound this value for pacing purposes. Userspace protocols on top of connected UDP can also use the interface, in which case it bounds the rate regardless of how quickly the process inserts data with `send` calls. Pacing requires a queuing discipline that supports the feature, like Fair Queue (FQ).

Batch size	CPU time %	Loss %
1	100	100
2	92	103
4	88	110
8	84	117

Figure 4: Netperf throughput as a function of send buffer size

Earliest Departure Time The socket option equates a socket with a flow to pace and thus works only for connected sockets. Protocols that multiplex flows over unconnected sockets, like QUIC, need a more fine grained mechanism. Kernel 4.19 acquired such a fine-grained time-based packet scheduling feature, `SO_TXTIME` [10] [11]. With this socket option, applications can pass a timestamp alongside data with control message `SCM_TXTIME`. The option encodes a 64-bit deadline in nanoseconds. Again, a qdisc has to be installed that understand the timestamp. The TBS qdisc is one option. FQ is another, though as of the time of writing it does not support the feature for unconnected sockets yet. The meaning of the deadline depends on flag `SCM_DROP_IF_LATE`. For the purpose of pacing, the flag is left disabled. In this mode, the timestamp is interpreted as the datagram's earliest departure time (EDT).

Kernel pacing saves cycles and decreases jitter compared to pacing in the application. Furthermore, `SO_TXTIME` was developed with full hardware offload in mind. A timing wheel [12] based approach to earliest deadline scheduling was previously presented in Carousel [13] in a software NIC. The SO_TXTIME patchset includes hardware offload support for the Intel igb driver.

Small flows Pacing limits individual flows' delivery rates. It works well for high rate flows, but the highly parallel server workload described in the introduction requires closer scrutiny. It is generally implemented at jiffy, or 1 millisecond, resolution (for HZ=1000). The example workload sends 1 MBps per connection, so one Kilobyte per interval. This is less than one segment, or below the threshold for achieving any batching benefits from segmentation offload.

One solution is to increase the interval, effectively allowing a slightly higher rate of bursting. Even without GSO this can noticeably reduce cycle cost through a reduction in timer interrupts. A preliminary investigation is shown in Figure 4. That compares CPU time and loss rate for various amounts of burstiness relative to a baseline without bursting. This was measured while sending normal datagrams. Both segmentation and pacing offload are disabled.

The relative loss rate is significant, but comes from a small base. These numbers hint that a segmentation offload with interval of 8 msec and `gso_size` of up to 8 may be a reasonable trade-off. Increase them both further and qualitative measurements of the video stream may start to be impacted.

Ideally, segmentation offload size and pacing can be configured fully independently. That will require a mechanism to specify and enforce different delivery times for constituent segments of a GSO datagram. This remains future work.

UDP GRO

Segmentation offload generates trains of packets. Which permits an analogous optimization on the receiver side, to consolidate multiple MTU sized packets into fewer larger ones. Large receive offload (LRO) does exactly this for TCP in hardware, and generic receive offload (GRO) in software.

The effectiveness of both depend on the length of packet trains. These are attenuated by pacing, as just discussed, as well as by the vagaries of queuing in the network. High rate, short distance use cases have a high chance of temporal locality. Due to batching within the base station, so do wireless connections. These environmental constraints apply equally to UDP as TCP. The benefits to WAN-facing servers are less clear and require more experimental data.

If carefully implemented, the combination of GRO and GSO reconstructs the exact segments as they arrived. This reduces protocol stack traversal with no discernible difference on the wire – apart possibly from packet (s)pacing, but that is affected by any router, segmenting or not. Generic receive offload is the missing piece of the stack. It is under active development as of writing.

Local Delivery Transparent delivery to a local socket requires breaking large payloads up back into their discrete segment sizes before passing to `recv`. Delivering one datagram on each `recv` call can be implemented by carving off `gso_size` from a GRO skb at the head of the receive queue. This complicates an already complex UDP receive path (e.g., with peek at offset) that has had its fair share of non-trivial race conditions. An alternative is to reuse the transmit segmentation path: segment the large packet just before enqueue. This clearly adds more overhead, not in the least from allocation of each of the segment skbs. It is not automatically a win over simply receiving each datagram without GRO, then. Even so, preliminary results are encouraging.

Transparent delivery is not necessarily the goal, though. More cycle efficient than either implementation is to send up the large payload as a whole. Paolo Abeni recently sent a second version of his RFC patchset that implements this approach [14] while being backward compatible with sockets that do not support GRO. It performs a socket lookup at GRO, similar to UDP tunnels, and performs GRO only if a socket has registered as GRO-capable. Sockets register interest in receiving coalesced datagrams with setsockopt `SOL_UDP/UDP_GRO`. If such a socket is found during GRO, the coalesced packet is sent to the process as is, with a control message to signal `gso_size`.

The socket lookup can be avoided if the segmentation approach proves cheaper than regular send. This requires more experimental data. One variant that may reduce segmentation cost is building chains of regular skbs at GRO time instead of a single skbuff that consists of page fragments. These are much simpler to segment later on, as they are just a linked list of segment-sized items. Steffen Klassert has presented such a solution [15].

These GRO optimizations are on top of another recent patchset by Edward Cree that independently increases receive path throughput by up to 25% [16]. That *listify* patchset also builds chains of skbs. It then calls the various stages of the receive path on a chain of packets at a time instead of traversing the entire path for each packet in order. This increases instruction locality and thus i-cache hitrate. Unlike for GRO, the listified skb chains need not take the exact same path, let alone have the same destination socket. The features are complementary.

	Gbps	calls/s	Mcycles/s	Speed-up (%)
UDP	798	568000	3564	100
UDP gro	1022	40250	2498	182

Figure 5: Cycle cost of `udpgso_bench_rx`

Evaluation Figure 5 shows very preliminary numbers from an earlier UDP GRO RFC patchset [17]. The number of system calls –and with that receive stack traversals– is reduced 14-fold while serving 1.28x more traffic. When looking at cycles and taking into account the higher delivery rate, efficiency is up 1.8x. We do need to bear in mind that these results are obtained under ideal circumstances, where UDP GSO generates a train of packets that arrives at the receiver in order, without drops, delays or interleaving packets. Again, conditions across the WAN are unlikely to be as favorable. Also, this test purposely avoids copying the data, which can dwarf the cost of the receive path. On the other hand, the 14x reduction in calls is lower than the 45x theoretical max.

QUIC Server Architecture

How do these optimizations affect production servers? We started this paper with a representative workload, a video server streaming tens of thousands of moderate streams at a time. This QUIC server spends a significant portion of its time in UDP networking.

The legacy architecture, previously described in detail [2], optimizes for cycle efficiency at the cost of non-trivial complexity. For reception, it employs a hybrid path of packet and udp sockets. Reading datagrams early in the receive path from packet sockets proved about 10% cheaper than going through the full protocol stack. This is with `PACKET_RX_RING` and a non-public `PACKET_INTERCEPT` patch that drops the packet in the kernel once accepted by a packet socket. A modern stack would use the newer `AF_XDP` sockets for this purpose. The packet sockets have a BPF filter that explicitly excludes fragments: IP defragmentation is a non-trivial, security sensitive task [18], so not duplicated in userspace. These fragmented packets are passed through the kernel IP stack to UDP sockets. Both packet sockets and UDP sockets are parallelized as their peak load can exceed a single CPU core's capacity. A rule of thumb is one socket per core. Traffic is split between packet sockets with BPF fanout, UDP sockets use `SO_REUSEPORT` BPF. A benefit of BPF is that selection can be based on QUIC connection ID, which remains stable even with connection hand-off between networks. All these features require superuser privileges, at least on process start-up.

The transmit path is slightly less complex. It also eschews UDP sockets. In this case, in favor of `SOCK_RAW` sockets.

These offer much of the benefit of packet sockets in terms of both performance and protocol configuration such as DSCP, without the complexity of having to maintain ARP and ND state in userspace. Experiments showed that the route lookup cost was not significant enough – even though the routes are not cached in this case. Packet sockets with `PACKET_TX_RING` were not significantly cheaper. Multiplexing many flows over few raw or unconnected sockets precludes offloading of pacing to the kernel with `SO_MAX_PACING_RATE`. That works only for connected sockets, where a socket implies a single flow and rate. As a result the server paces in userspace, with the jitter and cycle cost that entails.

This design is for the pure Linux kernel transmit path. It does not take into account a variant with the software NIC described in the Carousel paper [13], which uses shared memory and page pinning. Such bypasses are out of scope and, with a competitive kernel path, not needed.

New Design Some of the presented interfaces are very new. The new server architecture, as a result, is also a work in progress. We do not have detailed application level performance numbers at this point. Given the 2x transmit improvement in UDP GSO microbenchmarks and this workload being strongly skewed towards transmit, we do expect that we can do away with the entire complicated scaffolding in both receive and transmit paths and see a major reduction in cycles/Mbps.

The new design uses exclusively UDP sockets for both receive and transmit. It no longer requires superuser privileges. Receive sockets continue to use reuseport. Send sockets are still unconnected to save socket overhead at tens to hundreds of thousands of flows. Pacing is offloaded to the FQ qdisc with `SO_TXTIME`. Sufficiently large payloads are passed with `MSG_ZEROCOPY` flag.

As pointed out before, the goals of pacing many low rate flows at msec resolution and building large datagrams for GSO and zerocopy are somewhat at odds. The most pressing future work is fine-grained pacing of constituent datagrams within GSO segments. Hardware crypto offload is another. Finally, it is worth investigating whether more than 64KB can be passed per call, further increasing gains from zerocopy. This would imply building more than one GSO datagram per call.

Summary

Taken together, segmentation offload, zerocopy and pacing can considerably decrease cycle cost of serving content over UDP to make it approximate the efficiency of TCP, which has long had these features. UDP GSO shows a comparable gain to TCP GSO, near 2x. Smooth delivery with pacing lowers cpu overhead compared to userspace pacing and improves goodput by lowering drops. Zerocopy further avoids overhead from copying, 13% in the example workload. Full application results are still out, but we are confident that these improvements allow us to serve content with QUIC at considerably lower cycle overhead using a standard UDP datapath, compared to the earlier privileged packet socket based stack.

Acknowledgments

The end-to-end optimization of the UDP stack involves work from many people besides the authors. We have already mentioned Alexander Duyck's work on UDP and partial GSO [7] and ixgbe NIC support [4]. Boris Pismenny submitted UDP GSO support to the Mellanox mlx5 driver [5] and presented that work at the Netdev 0x12 conference [6]. SO_TXTIME is the work of Jesus Sanchez-Palencia [10]. Paolo Abeni implemented UDP GRO [14] and before that has made many performance improvements up and down the UDP stack. Steffen Klassert has presented a generic GRO mechanism based on frag_list that can also be applied to UDP [15]. Edward Cree implemented skb listification independent of GRO [16]. The authors also want to thank Ian Swett, Grzegorz Calkowski and Bin Wu at Google for adapting the Google serving infrastructure to these features and evaluating them. That work is ongoing.

References

[1] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. SIGCOMM '17. ACM, 2017.

[2] Jana Iyengar and Ian Swett. Quic: Design and internet-scale deployment. talk at netdevconf 0x12, Montreal, July 13th, 2018, 2018. https://netdevconf.org/0x12/session.html?developing-and-deploying-a-tcp-replacement-for-the-web.

[3] Willem de Bruijn. udp gso. netdev email thread, April 26th, 2018, 2018. http://patchwork.ozlabs.org/project/netdev/list/?series=41202&state=

[4] Alexander Duyck. ixgbe/ixgbevf: Add support for udp segmentation offload. netdev email thread, May 4th, 2018, 2018. https://patchwork.ozlabs.org/patch/908396/.

[5] Boris Pismenny. net/mlx5e: Add udp gso support. netdev email thread, June 28th, 2018, 2018. https://patchwork.ozlabs.org/patch/936521/.

[6] Boris Pismenny. Udp segmentation offload. talk at netdevconf 0x12, Montreal, July 13th, 2018, 2018. https://netdevconf.org/0x12/session.html?udp-segmentation-offload.

[7] Alexander Duyck. Gso: Support partial segmentation offload. netdev email thread, April 11th, 2016, 2016. https://patchwork.ozlabs.org/patch/608629/.

[8] Willem de Bruijn. msg_zerocopy. netdev email thread, August 3rd, 2017, 2017. https://netdevconf.org/2.1/session.html?debruijn and https://lwn.net/Articles/726917/.

[9] Willem de Bruijn. udp: zerocopy. netdev email thread, April 17th, 2018, 2018. http://patchwork.ozlabs.org/patch/899630/.

[10] Jesus Sanchez-Palencia. Scheduled packet transmission: Etf. netdev email thread, July 3rd, 2018, 2018. https://lwn.net/Articles/758592/.

[11] Jonathan Corbet. Time-based packet transmission, 2018. http://lwn.net/Articles/748879/.

[12] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87. ACM, 1987.

[13] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17. ACM, 2017.

[14] Paolo Abeni. udp: implement gro support. netdev email thread, October 19th, 2018, 2018. https://lwn.net/Articles/768995/.

[15] Steffen Klassert. http://patchwork.ozlabs.org/project/netdev/list/?series=65305, 2018.

[16] Edward Cree. Handle multiple received packets at each stage. netdev email thread, July 2nd, 2018, 2018. http://patchwork.ozlabs.org/project/netdev/list/?series=53249&state=*.

[17] Willem de Bruijn. udp and configurable gro. netdev email thread, September 14th, 2018, 2018. http://patchwork.ozlabs.org/project/netdev/list/?series=65763&state=*.

[18] NIST. Cve-2018-5391. CVE-2018-5390, 2018.

Appendix A: Test Setup

The results were obtained on a pair of dual-socket machines with two Intel Xeon E5-2696 18-core, 36-hyperthread chips each, connected over 10Gbps Ethernet using Mellanox ConnectX-3 NICs.

The kernel was davem-net-next/master including v4.19-rc8. Each machine was booted with `idle=halt` to reduce jitter, had a single network queue pair (`ethtool -L eth0 rx 1 tx 1`) with interrupts pinned to the same core as the test program, cpu 5.

The test programs are in the kernel sources at `tools/testing/selftests/net/udpgso_bench_(rt)x`.
The transmit command was run as

```
./udpgso_bench_tx -C 5 -4 -D ${DST_IP} -l 5 ${OPT}
```

where options are `-t` for TCP, `-z` to enable copy avoidance, `-c` for cache thrashing mode and `-S` for UDP GSO. The receive command is

```
./udpgso_bench_rx -C 5 ${OPT}
```

where options are `-t` for TCP and `-g` for UDP GRO. The tests were evaluated with `perf-tools`, using

```
perf stat -a -C 5 -e cycles -- ${CMD}
```

to measure cycles and

```
perf record -a -C 5 -e cycles -- ${CMD}
```

to record function traces. Segmentation offload was configured with `ethtool -K tso $TSO gso $GSO` after reverting commit 0a6b2a1dc2a2 ("tcp: switch to GSO being always on").