Google Cloud

A Brief Look at Network Performance Limiters

Not All Mbit/s Are the Same

Rick Jones

Disclaimer: In no way, shape, or form should the results presented in this document be construed as defining an SLA, SLI, SLO, or any other TLA. The author's sole intent is to offer helpful examples to facilitate a deeper understanding of the subject matter.

Introduction

Network Interface Card (aka NIC) speed is one of the most commonly understood limits to network performance. Coupled with a desire to demonstrate "Hitting link rate" this has led to using Mbit/s (or Gbit/s or ...) to report performance. However, there are many additional factors which can influence and limit network performance. This write-up will attempt to describe a few of those at a somewhat high level. It cannot and should not be construed as an exhaustive look, simply something to give an idea. For this write-up, instances in Google Cloud were used, with 8896 byte Guest/VPC MTU support configured, using same-region, same virtual subnet, VM to VM Internal IP communication.

It Is Not the Bits Which Matter But How They Are Packaged Which Counts

Transport protocols (eg. TCP, UDP, etc) seek to transfer data, bytes, from one place to another on behalf of their users. Regardless of the semantics they provide to the user, whether a byte stream from TCP or discrete messages from UDP, they accomplish this by bundling some quantity of the user's data into packets and sending them on their way. At the other end, they receive these packets, unbundle the data and present it to the receiver.

The process of packetizing a user's data and sending it on its way can be thought of as having two main cost/overhead types - per-packet and per-byte. Per-packet costs, as the name suggests, are costs for each packet. These are independent of the size of the packet. They include but are not limited to allocating buffers, adding headers, perhaps looking-up a route, passing the packet to the next "protocol" in the stack - eg TCP to IP to NIC driver to NIC, notifying the NIC about packets to send, or the NIC interrupting the system to tell it there are packets to receive. The more or fewer packets one sends/receives the more or less of that cost is incurred.

Per-byte costs are those costs accrued for each byte of data being sent/received. These are independent of the number of packets. They include copying data between user and networking, and often include computing some sort of checksum to provide statistical assurance only uncorrupted data will be given to the receiver. The more or fewer bytes one sends the more or less of that cost is incurred.

So, if we wish to send 1 MiB of data from one place to another, the per-byte cost of that will be virtually the same regardless of the number of packets used to send it, but the total per-packet costs will be quite different based on the number of packets used to do it.

Summary

What the table shows are the averages for Throughput (higher is better) and Service Demand[1] (lower is better) for both 1024 and 8192 bytes of data per send/packet, across UDP, TCP with stateless offloads on, and TCP with stateless offloads off. The TCP Maximum Segment Size (MSS) was set to match the send size to have the "on the wire" bytes per packet match between the protocols.

Throughput (Mbit/s - Higher is Better)			
Send/Packet Size	UDP	TCP on	TCP off
1024	4805	10638	4430
8192	18314	24820	19902
Improvement	3.81	2.33	4.49
Send Service Demand (usec CPU/KiB - Lower is Better)			
Send/Packet Size	UDP	TCP on	TCP off
1024	1.69	0.79	2.40
8192	0.43	0.32	0.42
Improvement	3.90	2.47	5.70
Receive Service Demand (usec CPU/KiB - Lower is Better)			
Send/Packet Size	UDP	TCP on	TCP off
1024	2.80	0.33	2.93
8192	0.63	0.28	0.56
Improvement	4.44	1.17	5.21

While going from 1024 bytes to 8192 bytes is an 8X improvement in bytes per packet and so packets per unit transferred, we do not see 8X improvement in throughput or service demand because there remain other constraints such as the per-byte costs, which remain essentially constant across the cases.
Computing improvement from the stateless offloads is left as an exercise for the reader :)

[1] Quantity of CPU consumed per unit of work.

Specifics

Consider the following set of netperf results where we flip at random between two different send sizes: 1024 and 8192 bytes in a UDP_STREAM test. With a 8896 byte VM vNIC MTU neither will require IP fragmentation[2]:

```
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to mongo.c.mumble.internal () port 0
AF_INET : histogram : spin interval : demo
```

Local Send Size	Local Send Throughput	Remote Recv Throughput	Local Send Calls	Remote Recv Calls	Local CPU Util %	Local Peak Per CPU Util %	Local Service Demand	Remote CPU Util %	Remote Peak Per CPU Util %	Remote Service Demand	Service Demand Units
1024	5369.66	5369.66	6554879	6554879	2.08	100.00	1.526	3.48	83.23	2.551	usec/KB
8192	24725.01	21091.06	3772828	3218318	2.02	96.73	0.321	3.25	100.00	0.516	usec/KB
1024	5391.05	5391.05	6580985	6580985	2.09	100.00	1.521	3.48	81.99	2.536	usec/KB
1024	5377.04	5377.04	6563879	6563879	2.08	100.00	1.524	3.89	92.04	2.842	usec/KB
1024	5223.88	5223.88	6376951	6376951	2.09	100.00	1.570	3.48	82.14	2.621	usec/KB
8192	24765.89	24765.89	3779038	3779038	2.03	97.31	0.322	3.11	53.50	0.493	usec/KB
1024	5328.60	5328.60	6504753	6504753	2.08	100.00	1.537	3.59	83.50	2.647	usec/KB
8192	24795.81	24795.81	3783637	3783637	2.01	79.32	0.319	3.01	57.11	0.478	usec/KB
8192	24692.77	24692.77	3767882	3767882	2.02	96.93	0.322	3.03	99.80	0.483	usec/KB
1024	5240.06	5240.06	6396762	6396762	2.09	68.43	1.565	3.84	90.75	2.883	usec/KB
1024	5352.77	5352.77	6534267	6534267	2.08	100.00	1.530	3.46	85.20	2.539	usec/KB
1024	5328.95	5328.95	6505232	6505232	2.08	56.30	1.537	3.54	86.59	2.615	usec/KB
8192	24423.61	21066.84	3726809	3214598	2.01	96.42	0.323	3.28	100.00	0.528	usec/KB
1024	5338.47	5338.47	6516901	6516901	2.09	61.00	1.536	3.86	94.26	2.843	usec/KB
8192	24404.63	24404.63	3723912	3723912	2.00	95.95	0.322	3.01	96.66	0.485	usec/KB
8192	24859.39	21036.38	3793448	3210071	2.03	70.73	0.321	3.28	100.00	0.519	usec/KB
8192	24651.22	24651.22	3761893	3761893	2.02	96.83	0.322	3.08	99.70	0.491	usec/KB
8192	24753.21	20967.21	3777108	3199400	2.02	96.93	0.321	3.31	100.00	0.525	usec/KB
1024	5335.27	5335.27	6512891	6512891	2.09	100.00	1.537	3.85	91.23	2.834	usec/KB
8192	24798.66	24798.66	3784202	3784202	2.02	96.92	0.320	3.10	65.23	0.492	usec/KB

The VMs used here were n2-standard-48s running Ubuntu 20.04 with a 5.11.0-1018-gcp Linux kernel and used the virtio_net vNIC driver. Their sysctl settings were at default values save for the receiving VM, which had *net.core.rmem_default* set to ~2 GiB in a not-entirely-successful bid to ensure there was no packet loss from overflowing the UDP socket receive buffer. We know that was occasionally effective because the number of "Remote Recv Calls" made by

[2]
```
HDR="-P 1"; for i in `seq 1 20`; do MESSAGE=1024; if [ $RANDOM -le 16384 ]; then MESSAGE=8192; fi;
netperf $HDR -H mongo -t UDP_STREAM -c -C -- -O
local_send_size,local_send_throughput,remote_recv_throughput,local_send_calls,remote_recv_calls,local_c
pu_util,local_cpu_peak_util,local_sd,remote_cpu_util,remote_cpu_peak_util,remote_sd,sd_units -m
$MESSAGE -M 64K,64K -R 1; HDR="-P 0"; done
```
Was used to collect the data

netserver occasionally equals the number of "Local Send Calls" made by netperf. When they were not the same, the losses were either at the receive socket buffer in the guest or in trying to get packets into the guest in the first place.

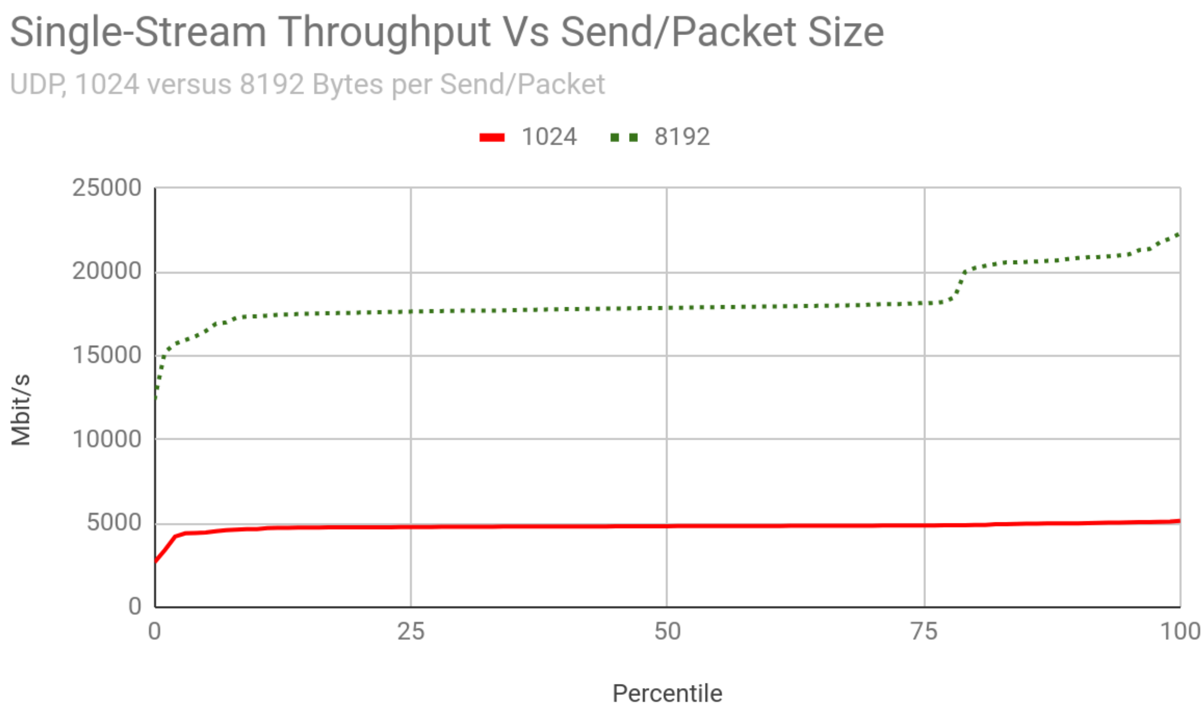
We asked netperf to report CPU utilization and service demand from both sides, along with the utilization of the most heavily utilized vCPU on either side. Service demand is a metric where netperf computes how much CPU time was consumed per unit of work. In this case it is the number of microseconds (usec) consumed per KB (really KiB[3]) transferred. Smaller is better.
So, what does it all mean? First of all notice that with the smaller send size, netperf was able to achieve about 5.3 Gbit/s. Notice also that while the overall ("Local CPU Util %") was low, at least one CPU was essentially saturated/pegged at 100% ("Local Peak Per CPU Util %"). In other words, a single flow/netperf, which will make use of the services of no more than one CPU (generally) was running as fast as that CPU could let it. You can also see that about 1.5X usec of CPU was consumed for every KiB of data sent by netperf ("Local Service Demand").

Over on the receiver, we did not peg any individual CPU, but the overall CPU utilization was higher, and so too then the service demand, at roughly 2.5 to 2.8 usec/KiB.
Contrast that with when the send size was 8192 bytes. Now the throughput is in excess of 24 Gbit/s, and the service demands are significantly lower. We were making fewer trips up and down the protocol stack for each KB of data transferred, and that is reflected in the CPU utilization and service demand.
A Mbit/s with 8192 byte messages was much less overhead than a Mbit/s with 1024 byte messages.

Not all Mbit/s are the same.

Of course, staring at a bunch of numbers can rapidly devolve into an exercise in eyestrain, so let's look at some pictures. First-up, UDP throughput from a second, much larger set of results than the above:

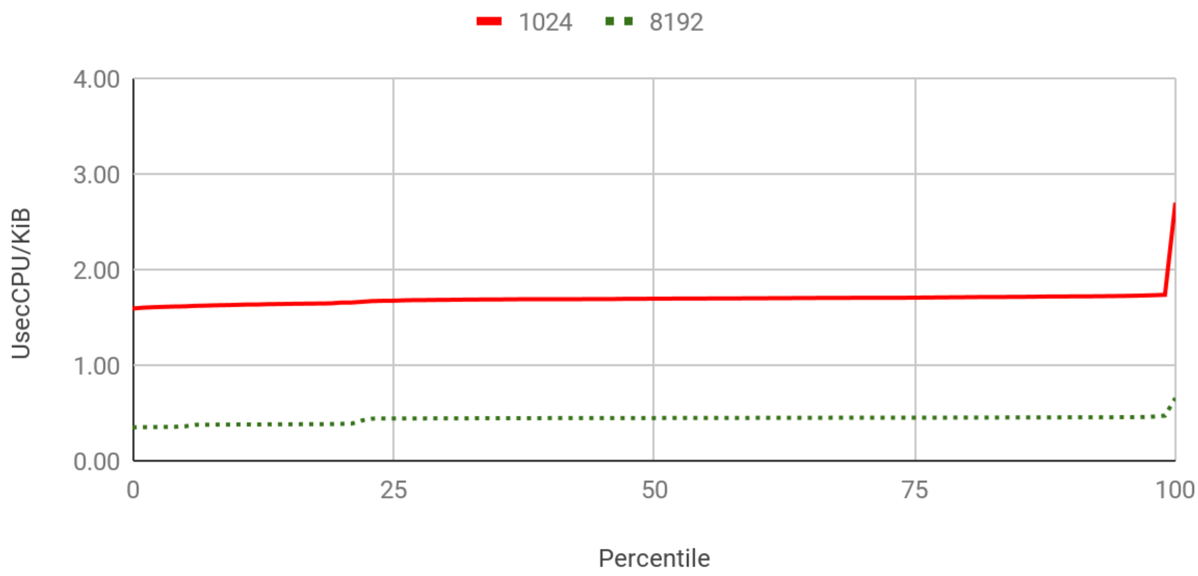
[3] netperf predates the broad adoption of the alternate names for Kilo, Mega, Giga, etc undertaken to make the SI unit folks content.



You can see in no uncertain terms how being able to send eight-times more data per packet results in significantly greater throughput. The benefits of greatly reducing the per-packet component of the overhead by greatly reducing the number of packets. Next, let's look at the service demands. Remember that with service demand, lower is better:

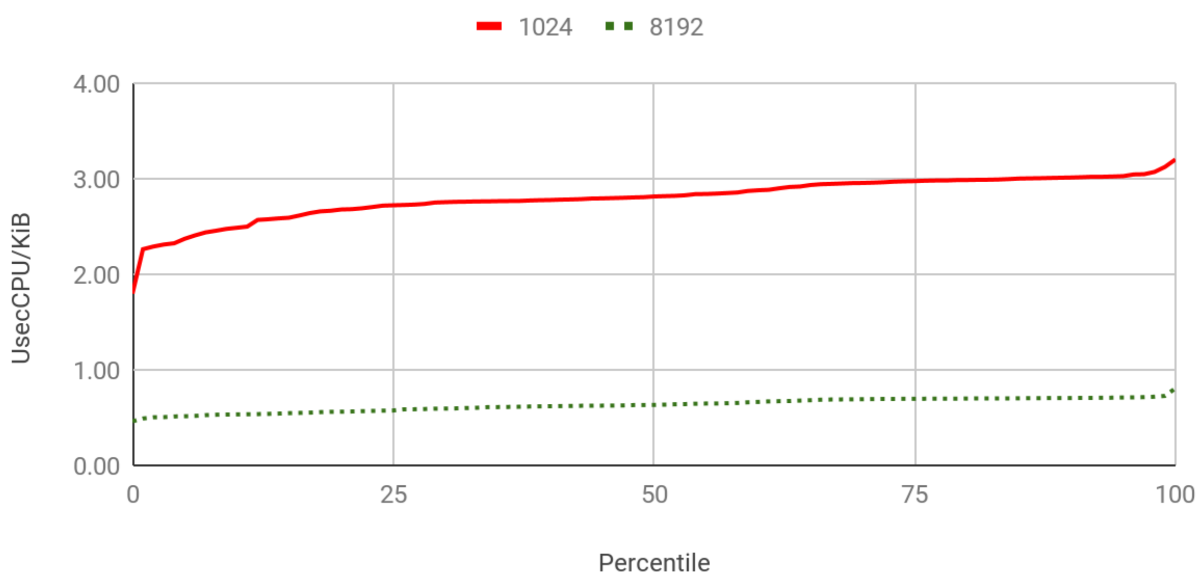
Single-Stream Send Service Demand Vs Send/Packet Size

UDP, 1024 versus 8192 Bytes per Send/Packet



Single-Stream Receive Service Demand Vs Send/Packet Size

UDP, 1024 versus 8192 Bytes per Send/Packet



Not all Mbit/s are the same.

So Why Do I Get High Throughput With TCP Even On A Network with an MTU of 1500 bytes?

In a phrase, “Stateless Offloads.”

Stateless offloads are offloads to the NIC (or what the networking stack perceives as the NIC) which do not require the NIC to retain persistent state for connections/flows. NIC vendors started to provide these in the mid to late 1990s to help lessen the overheads systems had to endure as the NIC bit rates increased at a rate faster than per-core CPU horsepower while the IEEE refused to sanction larger MTUs for Ethernet.

The first of these was checksum offload. Checksum computation and validation is a per-byte cost. Offloading it to the NIC freed-up about 10% of the CPU cycles a stack would consume at the time processing bulk-transfer traffic. To transfer a given quantity of data, there still must be as many trips up and down the protocol stack as before.

With the advent of checksum offload, two additional offloads became feasible. The first was TCP Segmentation Offload (aka TSO). From one TCP data segment to the next, the only two things which tend to change in the TCP header are the sequence number and the checksum. [4] With TCP segmentation offload, the system’s networking stack can hand the NIC a large quantity of data (eg. 64 KiB), an initial TCP/IP header template, and the Maximum Segment Size for the connection, and the NIC can then create the TCP segments for the stack, filling-in those header fields which have to change.

Later, a receive side version of this was created called GRO or Generic Receive Offload. Often this is done at a very low level in the receiver’s networking stack, but some NICs implement it as well.[5] The virtio_net vNIC implementation in GCP initially provided it as “LRO” (Large Receive Offload). Although starting around the 5.13 Linux kernel, control of that shifted to “GRO-HW” - basically Generic Receive Offload implemented in HardWare.

You can see how this would have an effect similar to having a larger MTU and being able to send larger packets. The per-packet costs of going down (and up) the protocol stack on either

[4] Yes, there can be TCP timestamps, and we are ignoring the ID field of the IP datagram header but the NIC can deal with those too :)

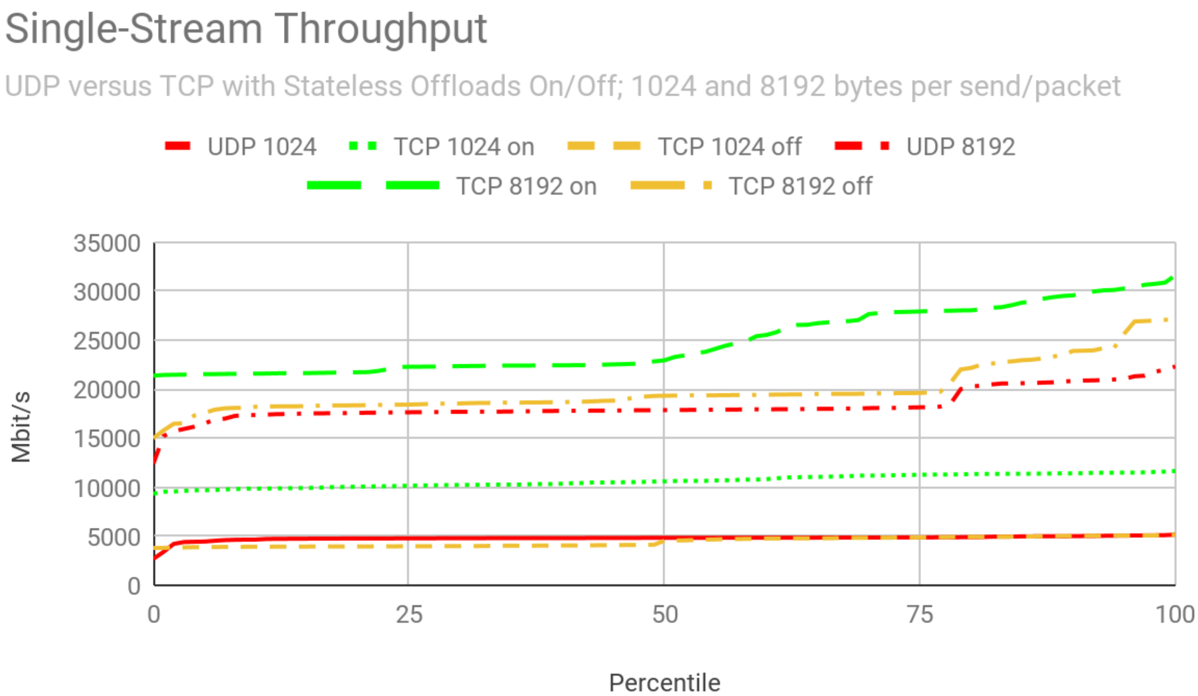
[5] There is a whole history here involving Generic Segmentation Offload (GSO - NIC-independent version of TSO) and a NIC-based receive offload called Large Receive Offload (LRO) that isn’t really germane to the discussion.

side are *drastically* reduced for a given quantity of data. One wag, who shall remain nameless, even dubbed TSO “Poor Man’s Jumbo Frames.”

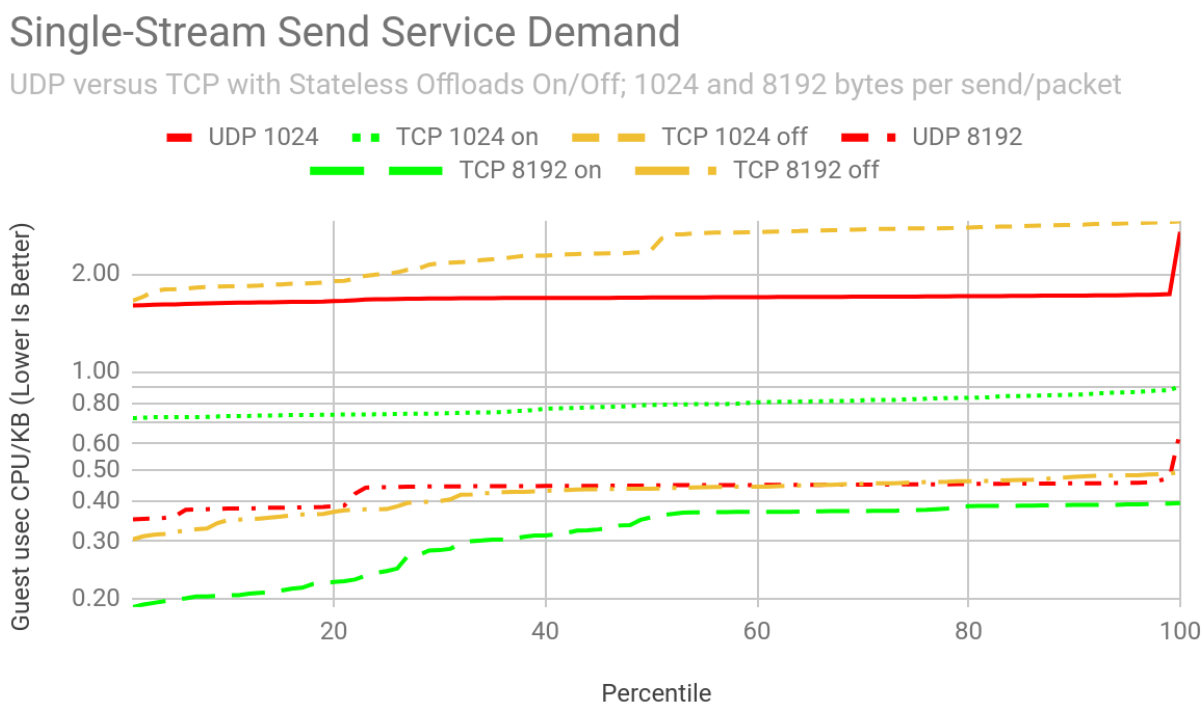
Let’s look at TCP_STREAM now between those two instances, where we alter the TCP Maximum Segment Size to have each TCP segment “on the wire” carry as many bytes as was carried in each UDP datagram above. To keep things further “even” between the two, we will also put as many bytes into each send() call as with the UDP test - ie 1024 or 8192 bytes. The instances have stateless offloads enabled:

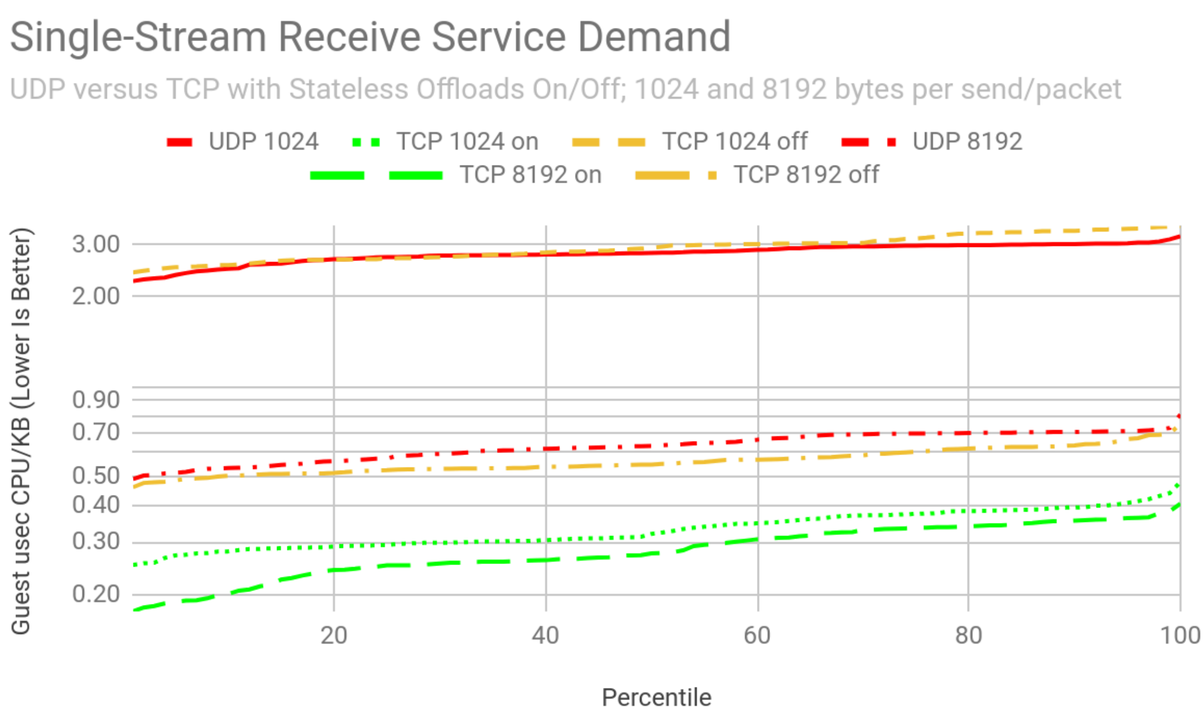
Transport MSS bytes	Throughput	Local CPU Util %	Local Peak Per CPU Util %	Local Service Demand	Remote CPU Util %	Remote Peak Per CPU Util %	Remote Service Demand	Service Demand Units
8192	29057.83	1.34	63.47	0.181	1.41	55.96	0.191	usec/KB
1024	11475.16	2.11	100.00	0.723	0.87	37.93	0.300	usec/KB
1024	11093.31	2.12	99.90	0.752	1.07	42.30	0.380	usec/KB
8192	31730.05	1.53	69.94	0.190	1.55	35.96	0.192	usec/KB
1024	11364.59	2.12	100.00	0.733	1.08	28.02	0.373	usec/KB
8192	30099.32	1.44	67.27	0.188	1.47	45.73	0.191	usec/KB
1024	12002.34	2.10	100.00	0.688	0.93	39.23	0.306	usec/KB
1024	11224.30	2.14	100.00	0.748	0.98	31.21	0.345	usec/KB
8192	31473.70	1.73	54.18	0.216	1.81	69.67	0.226	usec/KB
1024	11141.19	2.10	99.90	0.743	0.88	32.35	0.312	usec/KB
1024	11179.73	2.11	100.00	0.743	0.90	41.21	0.317	usec/KB
1024	11112.34	2.11	93.70	0.746	0.87	32.23	0.308	usec/KB
1024	11854.08	2.11	100.00	0.701	0.85	21.79	0.282	usec/KB
1024	11245.71	2.12	100.00	0.740	0.95	29.90	0.332	usec/KB
8192	31054.90	1.72	55.56	0.218	1.74	43.43	0.220	usec/KB
1024	11942.22	2.11	100.00	0.695	0.92	34.98	0.304	usec/KB
8192	28080.53	1.47	58.88	0.206	1.74	37.04	0.244	usec/KB
1024	11952.95	2.09	99.80	0.688	0.96	43.56	0.315	usec/KB
1024	11857.43	2.09	100.00	0.695	0.93	42.68	0.308	usec/KB
8192	28674.30	1.33	58.03	0.182	1.55	38.98	0.212	usec/KB

We can see a considerable difference in the service demands between the TCP and UDP cases. Even with the small “on the wire” packet sizes, the stateless offloads (LRO/GRO and TSO/GSO) allow TCP to achieve markedly higher throughput. Disabling those stateless offloads narrows the gaps considerably, and can even lead to UDP being “faster” than TCP. Let’s look at some comparison charts starting with the 1024 byte send/packet size:



You can see that with the stateless offloads either not applicable (UDP) or "off" TCP and UDP are performing very roughly at the same level for both 1024 and 8192 byte sends/user bytes per packet. And that stateless offloads bump TCP's performance considerably. The reason being it greatly reduces the overhead to send and/or receive data, as can be seen in the next two charts, showing service demands:





Again, we can see how either larger packets on the wire, or the stateless offloads (TSO/GSO, LRO/GRO) greatly improve the per-unit cost of data transfer.

Not all Mbit/s are the same.

Sometimes the Limit is Packets Per Second

In at least a few of the previous examples, we've saturated one (or perhaps more) CPUs in our systems. Coupled with the other tests, we have seen how the number of packets/s to achieve a given Mbit/s matters. To help reinforce that, let's run another set of UDP tests with our systems, this time using a much larger number of message sizes.[6]

```
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to large () port 0
AF_INET : histogram : spin interval : demo
Local Local      Remote     Local    Remote
Send  Send       Recv       Send     Recv
Size  Throughput Throughput Calls    Calls

1     5.56       4.49       6950708  5610243
2     11.04      10.60      6903261  6627224
3     16.60      16.41      6916297  6836581
4     21.95      21.78      6859230  6806028
16    87.06      82.20      6801488  6421962
32    178.26     178.19     6963426  6960609
64    354.48     326.29     6923413  6372838
128   691.31     663.44     6751108  6479015
256   1351.93    1351.93    6601352  6601352
512   2555.18    2345.51    6238309  5726423
1024  4782.00    4782.00    5837477  5837477
2048  8327.85    8327.85    5082989  5082989
4096  12937.09   12937.09   3948198  3948198
8192  18356.68   18356.68   2801069  2801069
```

Notice how for message sizes 1 (one) through 64 bytes the number of send calls is roughly the same while the send throughput was increasing. For these runs, our performance was packet-per-second limited rather than Mbit/s. 128, 256, and 512 byte messages were somewhat close in packets per second. Beyond that things beyond packet per second limits started to take the fore.

Not all Mbit/s are the same.

[6] Yes, stateless offloads are re-enabled :) Though given the only one which applies to UDP is checksum offload, and we never disabled that, it doesn't really matter.

What about at a constant bitrate?

For the first UDP test we were not holding the bitrate constant. We were letting netperf send as fast as it could. So, to address that, let's run UDP_STREAM again, but tweaking the netperf command lines to send at the same bit rate for each message size. Rather than use netperf's built-in pacing, we instead push the 'fq' qdisc onto the egress interface and have netperf tell it what rate we want. Since 'fq' is considering the entire packet, headers and all, and we want 1 Gbit/s at the user level, we adjust what we tell 'fq' accordingly[7].

```
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to large.c.mumble.internal () port 0
AF_INET : histogram : spin interval : demo
Local Local      Remote     Local   Remote  Local Local   Local   Remote Remote  Remote  Service
Send  Send       Recv       Send    Recv    CPU   Peak    Service CPU    Peak    Service Demand
Size  Throughput Throughput Calls   Calls   Util  Per CPU Demand  Util   Per CPU Demand  Units
                                            %     Util %          %      Util %
8192  1000.08    1000.08    152604  152604  0.12  5.75    0.471   0.19   7.32    0.747   usec/KB
1024  1000.04    1000.04    1220773  1220773  0.50  23.93   1.968   0.69   19.91   2.731   usec/KB
8192  999.77     999.77     152556  152556  0.12  5.74    0.471   0.17   6.57    0.674   usec/KB
8192  1000.07    1000.07    152602  152602  0.13  6.23    0.510   0.15   4.15    0.570   usec/KB
1024  1000.02    1000.02    1220789  1220789  0.50  23.90   1.958   0.50   18.86   1.963   usec/KB
1024  1000.02    1000.02    1220756  1220756  0.49  23.48   1.924   0.44   11.42   1.737   usec/KB
1024  1000.02    1000.02    1220773  1220773  0.50  13.85   1.954   0.51   21.53   2.010   usec/KB
8192  1000.07    1000.07    152602  152602  0.13  6.13    0.503   0.18   7.63    0.691   usec/KB
8192  1000.08    1000.08    152604  152604  0.13  4.50    0.509   0.17   5.54    0.651   usec/KB
1024  1000.00    1000.00    1220722  1220722  0.50  24.21   1.983   0.65   12.59   2.559   usec/KB
1024  1000.03    1000.03    1220773  1220773  0.50  23.72   1.951   0.56   21.59   2.212   usec/KB
8192  1000.08    1000.08    152604  152604  0.13  6.33    0.526   0.12   4.55    0.480   usec/KB
1024  999.98     999.98     1220705  1220705  0.50  23.79   1.949   0.62   12.38   2.427   usec/KB
1024  1000.03    1000.03    1220773  1220773  0.50  23.95   1.962   0.57   20.26   2.242   usec/KB
8192  1000.04    1000.04    152597  152597  0.13  5.85    0.495   0.16   6.38    0.629   usec/KB
1024  999.86     999.86     1220556  1220556  0.51  24.49   2.006   0.69   15.73   2.716   usec/KB
1024  1000.02    1000.02    1220755  1220755  0.50  23.79   1.949   0.70   29.37   2.759   usec/KB
8192  1000.07    1000.07    152602  152602  0.14  6.52    0.543   0.22   7.53    0.872   usec/KB
1024  1000.01    1000.01    1220772  1220772  0.50  23.84   1.953   0.48   16.88   1.900   usec/KB
8192  1000.04    1000.04    152597  152597  0.13  5.62    0.502   0.17   5.34    0.667   usec/KB
```

This makes the packetization difference that much more clear. At each message size we were sending essentially 1 Gbit/s (at the user-level, UDP/IP/etc headers not included). For the same

---

[7] Using the following: `HDR="-P 1"; for i in `seq 1 20`; do MESSAGE=1024; if [ $RANDOM -le 16384 ]; then MESSAGE=8192; fi; WHDRS=`expr $MESSAGE + 42`; RATE=`expr 125000000 * $WHDRS`; RATE=`expr $RATE / $MESSAGE`; netperf $HDR -H gve-mongo -t UDP_STREAM -c -C -- -q $RATE -O local_send_size,local_send_throughput,remote_recv_throughput,local_send_calls,remote_recv_calls,local_cpu_util,local_cpu_peak_util,local_sd,remote_cpu_util,remote_cpu_peak_util,remote_sd,sd_units -m $MESSAGE -M 64K,64K -R 1; HDR="-P 0"; done`

bitrate, 1024 byte messages needed 8X the number of packets as 8192 byte messages. And we can see the ~3-4x difference in the receiving service demand and a roughly 4X difference in sending service demand. It wasn't an 8X difference because not all the costs are per-packet. Even with checksum offload there are still data copies from the stack to the receiving netserver, aka per-byte costs.

Not all Mbit/s are the same.

Configuration

VMs

For these tests, a pair of n2-standard-48 instances were used, with the containing project allowlisted for the Private Preview of large MTU support. Private IP address communication was employed, with the two instances in the us-west3-a availability zone.
The distro used was Ubuntu 20.04, with the 5.11.0-1018-gcp kernel[8] and "virtio" vNICs driven by the virtio_net driver.

Script For Percentile Data

The following script was used to gather the data used in the percentile charts. Do not consider it an acme of scripting, it was quick and sufficient to the task at hand.

```
#!/usr/bin/bash

# assuming a pair of systems with an MTU >= 9000 bytes, perform a set of netperf
# tests to demonstrate the effects of stateless offloads and how it enables TCP
# to perform better than UDP

# some of these will be redundant or uninteresting for certain tests but we'll
# keep them for the sake of simplicity. both here and in later post-processing
OUTPUT="protocol,result_brand,local_send_size,transport_mss,local_send_throughput,remote_recv_throughput,local_send_calls,remote_recv_calls,local_cpu_util,local_cpu_peak_util,local_sd,remote_cpu_util,remote_cpu_peak_util,remote_sd,sd_units"

DESTINATION=$1
```

[8] Somewhere between the 5.11.0-1018-gcp and 5.11.0-1021-gcp kernels a functional regression took place where it once again became impossible to disable Large Receive Offload (LRO). So, if looking to reproduce these results, be certain to verify that disabling LRO does indeed work in your kernel(s) of choice. Disabling tcp-gro-hw seems to have become the way to go for that.

```
ITERATIONS=$2
DEVICE=$3

for i in `seq 1 $ITERATIONS`
do
  MESSAGE=1024
  if [ $RANDOM -ge 16384 ]
  then
    MESSAGE=8192
  fi

  PROTOCOL="udp"
  if [ $RANDOM -ge 16384 ]
  then
    PROTOCOL="tcp"
  fi

  if [ $RANDOM -le 16383 ]
  then
    STATELESS="on"
    sudo ethtool -K $DEVICE tso on gso on lro on gro on
    ssh $DESTINATION "sudo ethtool -K $DEVICE tso on gso on lro on gro on"
  else
    STATELESS="off"
    sudo ethtool -K $DEVICE tso off gso off lro off gro off
    ssh $DESTINATION "sudo ethtool -K $DEVICE tso off gso off lro off gro off"
  fi

  netperf $HDR -H $DESTINATION -t omni -c -C -B $STATELESS -- -T $PROTOCOL -d send -o
$OUTPUT -m $MESSAGE -M 128K,128K -R 1 -G `expr $MESSAGE + 12`
  HDR="-P 0"
done
```