Google Cloud

Considerations When Benchmarking UDP Bulk Flows

We do not think that result means what you think it means

Rick Jones, Andy Zhu

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 authors' sole intent is to offer helpful examples to facilitate a deeper understanding of the subject matter.

Introduction

While TCP (Transmission Control Protocol) is arguably the more "popular" protocol, there are applications which make extensive use of UDP (User Datagram Protocol) and so there will be times when one wishes to benchmark UDP performance rather than TCP. This write-up will attempt to describe some of the important considerations when running UDP-based benchmarks between systems. While the systems discussed in this write-up are instances (ie VMs) in Google Cloud, a public cloud[1], the ideas presented are not especially cloud-specific.

TL;DR: Just tell me what to look for and how

When presented with a situation where someone says that the UDP throughput across the network between their servers is bad and they are blaming the network, you should start by checking for problems in the servers/applications. When those systems are running Linux:

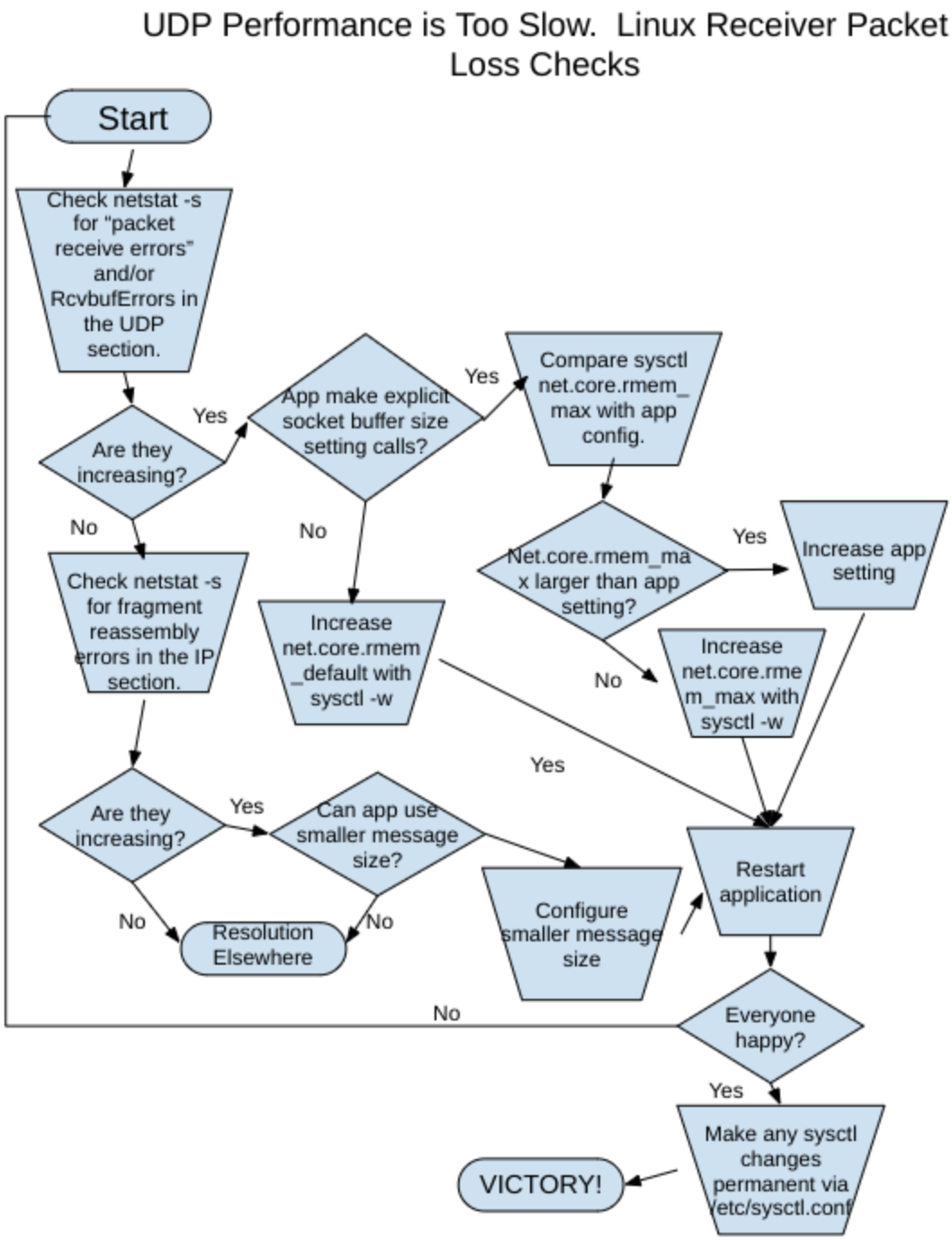
1. Check for "packet receive errors" and/or RcvbufErrors in the UDP section of netstat -s output on the receiver. If they are increasing, the receiving application likely needs a larger socket receive buffer. If the application makes explicit calls to set the socket receive buffer size, look to tweak net.core.rmem_max via sysctl -w and/or the application. If the application does not make explicit calls to set socket receive buffer size, look to tweak net.core.rmem_default via sysctl -w. 8MB is a reasonable starting point, but it may be necessary to go higher. The changes will not affect existing sockets. Edit /etc/sysctl.conf to make sysctl changes permanent. If there is still packet loss with a Very Large (™) socket receive buffer it suggests the sending side is simply faster than the receiving side and further increases in socket receive buffer size will not help. Proper flow control between sender and receiver is required in this situation.
2. On the sending side, check for SndbufErrors in the UDP section of the output of netstat -s. If they are increasing, the application likely has a socket send buffer large enough to hold more messages than the vNIC's txqueuelen and so defeat the intra-stack flow control. The txqueuelen is checked via ifconfig. If the application

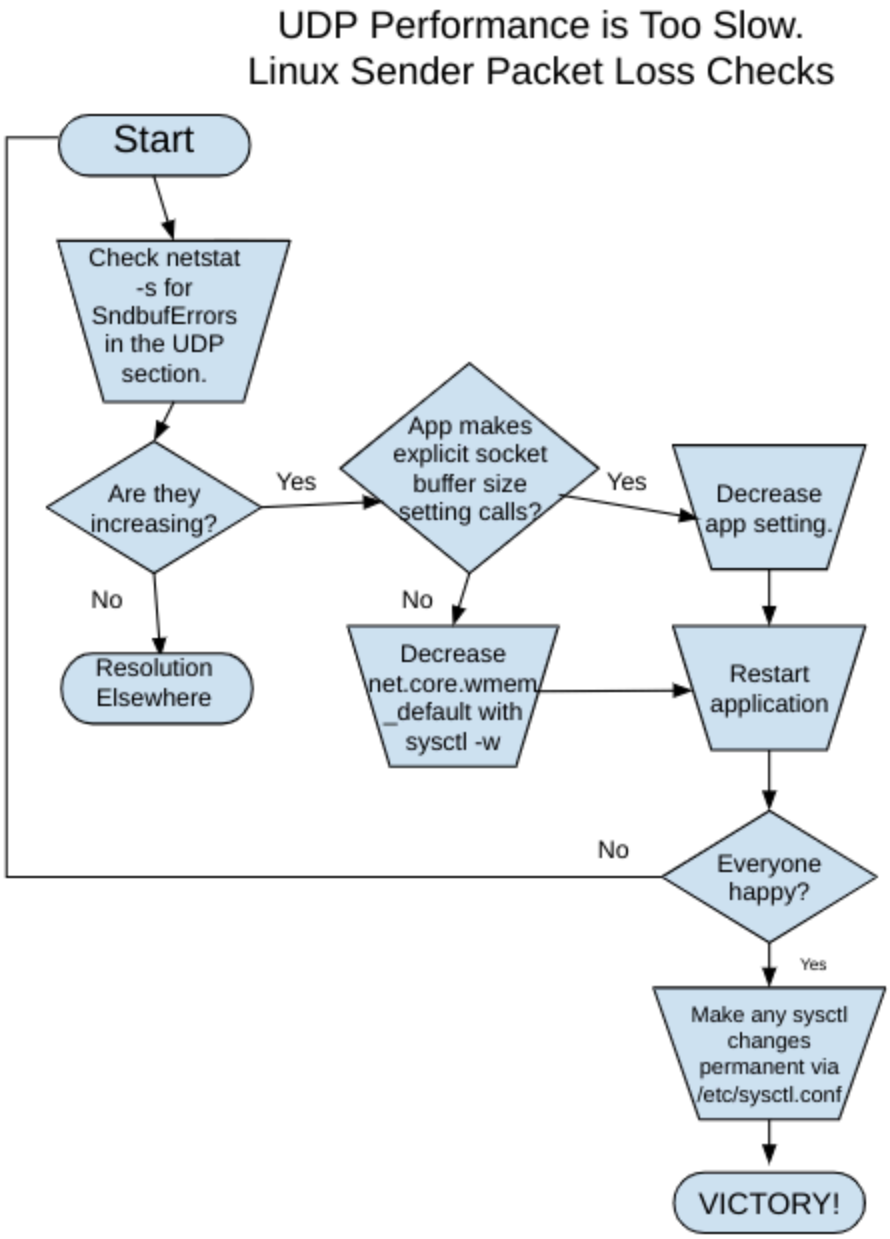
[1] Specifically, they were n2-standard-4-Icelake VMs running Ubuntu 20.04 in the us-central1-b zone of Google Cloud.

makes explicit calls to set the socket send buffer size, we suggest that the application use a value smaller than application_message_size*vNIC txqueuelen. If the application does not make explicit calls to set socket send buffer size, get the value of net.core.wmem_default with sysctl and compare it with application_message_size*vNIC txqueuelen. If it is larger, reduce the value of net.core.wmem_default with sysctl -w. The changes will not affect existing sockets. Edit /etc/sysctl.conf to make sysctl changes permanent.

3. Once those statistics no longer increment, the issue may reside elsewhere beside the systems, though the application may lack needed flow control or abuse IP fragmentation. On the receiving side, check the fragmentation statistics in the IP block of netstat -s output. If the failure and/or timeout statistics are incrementing, investigate whether the application can use message sizes which do not require IP fragmentation. Application message sizes smaller than the vNIC MTU less 28 bytes are best. The vNIC MTU will be included in the output of ifconfig, though sometimes the MTU may be misconfigured with an unsupported value.

A version of this in flowchart form for the receiving side and sending side are presented below:


UDP Performance is Too Slow. Linux Receiver Packet Loss Checks
Start
Check netstat -s for "packet receive errors" and/or RcvbufErrors in the UDP section.
Are they increasing?
Yes
App make explicit socket buffer size setting calls?
Yes
Compare sysctl net.core.rmem_max with app config.
No
Check netstat -s for fragment reassembly errors in the IP section.
No
Increase net.core.rmem_default with sysctl -w
Net.core.rmem_max larger than app setting?
Yes
Increase app setting
No
Increase net.core.rmem_max with sysctl -w
Yes
Are they increasing?
Yes
Can app use smaller message size?
No
Resolution Elsewhere
No
Configure smaller message size
Restart application
Everyone happy?
No
Yes
Make any sysctl changes permanent via /etc/sysctl.conf
VICTORY!


UDP Performance is Too Slow.
Linux Sender Packet Loss Checks
Start
Check netstat -s for SndbufErrors in the UDP section.
Are they increasing?
Yes
No
Resolution Elsewhere
App makes explicit socket buffer size setting calls?
Yes
No
Decrease app setting.
Decrease net.core.wmem_default with sysctl -w
Restart application
Everyone happy?
No
Yes
Make any sysctl changes permanent via /etc/sysctl.conf
VICTORY!

Summary

- **The default values** configured under Linux for socket receive buffer default and maximum size for UDP **are too small** for UDP bulk flow applications/benchmarking at any data rate besides the lowest transfer rates.
- Receive socket buffer sizes of 8MB or more are a reasonable starting point to avoid socket buffer overflows.
- One should pick a receive socket buffer size to account for the transfer rate, how often and for how long the receiving application might be held-off from reading from the socket(s) and validate that choice by examining the statistics for UDP socket buffer overflow on the receiving system.
- There are other sources of packet loss besides receive socket buffer overflow.
- It is possible to have too large a socket send buffer size.
- Your Mileage WILL Vary.

Let it Rip

At first, it might be tempting to just grab one's favorite networking benchmark and let it rip. This write-up will utilize netperf, but there are others. The common TCP case might look like:

```
sender$ src/netperf -t TCP_STREAM -H 10.138.0.2
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.2 () port
0 AF_INET : histogram : spin interval : demo
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
 131072  16384  16384    10.00    9742.91
```

It is pretty straight-forward - the test achieved ~9.7 Gbit/s, ran for 10 seconds and was writing 16384 bytes at a time into the socket[2]. Now let's let it rip with UDP_STREAM test instead:

[2] By default, on each sending call a netperf TCP_STREAM test will send as many bytes as was the size of the send socket buffer (SO_SNDBUF) at the time the data socket is created. There is "nuance" with the Recv and Send socket buffer sizes for TCP connections which is specific to Linux that we won't get into here except to say that those values displayed were not what the socket buffer sizes became by the end of the test. Regardless, netperf was sending 16384 bytes at a time into the TCP socket over the entire test.

```
sender$ src/netperf -t UDP_STREAM -H 10.138.0.2 -- -R 1
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.2 () port
0 AF_INET : histogram : spin interval : demo
Socket  Message  Elapsed      Messages
Size    Size     Time         Okay Errors   Throughput
bytes   bytes    secs            #      #   10^6bits/sec
212992   65507   10.00      175550      0    9199.75
212992           10.00      175320           9187.69
```

First off, there are two output lines. The first is for sending-side information. It is telling us that the send socket buffer size was 212992 bytes. That is the default size for a send socket buffer for UDP under Linux. In this case that is indeed what it was by the end of the test as well, as there is no nuance involved with UDP under Linux similar to what there is for TCP. We also see that the number of bytes being written into the socket each time is not the same as the TCP test - it is 65507 bytes. Why such an odd looking value? Well, that is netperf's default for a UDP_STREAM test. An IPv4 datagram can be no larger than 65535 bytes, IPv4 header included. A UDP datagram header is 8 bytes, and an IPv4 header is (virtually) always 20 bytes. 65535 - 28 is 65507 so that is the most one can send in one UDP datagram when using IPv4[3].

We are also told there were 175550 successful "Okay" send calls, and that the sending-side rate was ~9 Gbit/s. A little slower than the TCP case, but reasonably close.

That second line is information for the receiving side. Again, the socket buffer was 212992 bytes. This happens to be the default receive socket buffer size for UDP under Linux. We also see the number of successful message receives is 175320 - lower than the number of sends, and a correspondingly lower Gbit/s.

So, somewhere between the sending calls made by netperf, and the receiving calls made by the netserver, 230 messages were lost. Some folks, particularly those who hadn't just run a TCP test might assume that the network between the two instances was good for only about 9.2 Gbit/s. We know though that is not the case, because we've seen that network do in excess of 9.7 Gbit/s. So, what is going on?

Issue One: Flow Control

TCP provides end-to-end flow control which ensures a sender will "never" overrun a receiver. It will never send more data towards a receiver than the receiver has said it is able to take at

one time. TCP also employs congestion control to try to avoid overrunning points along the network between the two.

UDP provides no flow control.

UDP will send data just as fast as the application and the networking stack on the sending side will allow[4]. If the receiver happens to be a little bit slower than the sender, either just occasionally, or for the duration of the test, then the receiver's receive socket buffer will overflow, and messages will be dropped. All the packet processing in the stack to get the packet to the socket buffer is then wasted. And in fact, if a sender is sufficiently faster than a receiver, one can end up with a receive rate of virtually zero as the receiving side spends all its time just discarding packets.

Under Linux, statistics for UDP receive socket buffer overflows can be seen in "netstat -s" statistics. For example:

```
receiver$ netstat -s
...
Udp:
     351291 packets received
     12 packets to unknown port received
     448 packet receive errors
     347 packets sent
     448 receive buffer errors
     0 send buffer errors
...
```

We are interested in "packet receive errors" and specifically RcvbufErrors. Ninety-nine times out of ten (sic) when those increase it is for a receive socket buffer overflow. Now, if we run the UDP test again, and snap those statistics again on the receiver after the test completes:

```
sender$ src/netperf -t UDP_STREAM -H 10.138.0.2 -- -R 1
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.2 () port
0 AF_INET : histogram : spin interval : demo
Socket  Message  Elapsed      Messages
```

[3] Nuttcp, another networking benchmark, will default to 8192 bytes under Linux and 1024 bytes under Windows for UDP. Some versions of iperf3 will default to 8192 bytes for UDP. Others will pick a message size based on the TCP MSS of the control connection. Caveat Benchmarker!

```
Size    Size     Time         Okay Errors   Throughput
bytes   bytes    secs            #      #   10^6bits/sec
212992   65507   10.00      174498      0    9144.63
212992           10.00      174268           9132.57

receiver$ netstat -s | fgrep “receive buffer”
    676 receive buffer errors
```

This time around, 230 messages were lost. From the netstat statistics we can see there were 228 socket buffer overflows[5]. That means the vast majority of the losses were from situations where the receiving application (in this case the netserver process) was not always keeping up with the incoming traffic. Perhaps the netserver process got delayed in running for a little while. Perhaps the traffic became a bit bursty. Either way, 228 times an arriving 65507 byte message found the 212992 byte receive socket buffer too full to hold it. So, let’s try using a larger socket buffer and see what happens. In fact, just for the fun of it, let’s use an enormous socket buffer on the receiver. Later in this paper we will discuss ways to be more thoughtful when picking a value.

```
receiver$ netstat -s | fgrep “receive buffer”
    858 receive buffer errors

sender$ src/netperf -t UDP_STREAM -H 10.138.0.2 -- -R 1 -S 24M
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.2 () port
0 AF_INET : histogram : spin interval : demo
Socket  Message  Elapsed      Messages
Size    Size     Time         Okay Errors   Throughput
bytes   bytes    secs            #      #   10^6bits/sec
212992   65507   10.00      175992      0    9222.81
50331648 [6]          10.00      175988           9222.60

receiver$ netstat -s | fgrep “receive buffer”
    858 receive buffer errors
```

[4] Linux actually provides “intra-stack” flow control to (generally) keep a UDP sender from sending faster than the egress (sending) network interface, but that is not an end-to-end flow control mechanism. Further, if an application has an SO_SNDBUF size sufficient to hold more messages than can be queued to the driver that flow control will not be effective.
[5] These netstat statistics are system-wide. For our purposes we can assume that since we are the only non-trivial source of UDP traffic, any UDP receive socket buffer overflow is “ours.”

The addition of the test-specific "-S 24M" option to the command line causes netperf to tell netserver (the receiver) to make calls to set the socket buffer size to 24*1048576 bytes. This time there were no receive buffer overflows!

Issue Two: IP Fragmentation

TCP will go to some pains to avoid sending TCP segments requiring fragmentation by IP. Essentially all UDP does is slap a UDP header on the user's data and hand it to IP to deal with. Netperf's default send size for UDP_STREAM is 65507 bytes, yielding a 65535 byte IPv4 datagram. The IP MTU of the network interfaces of the instances being used for this write-up is 1460 bytes. 65535 is certainly larger than 1460, which means the IPv4 datagrams carrying the UDP datagrams carrying netperf's 65507 bytes of data will have to be fragmented. In this case they will be fragmented into 45 IPv4 datagram fragments. Those fragments will be sent across the network and will be reassembled at the receiver.

Why is this a problem?

That is a problem because **all** the fragments must arrive at the receiver to reassemble the full datagram. If any fragments are lost, the entire datagram is as good as lost and the entire datagram must be retransmitted by the upper layer protocol.

If we wave our hands a bit about the reasons for packet losses and how they are distributed, and assume the network has a packet loss probability we will refer to as 'p' then the probability of any one packet not being lost is (1-p). Let's call this 'P'. So, the probability of all 45 of our fragments of any one datagram making it across the network is P^45. The probability of any of our messages being lost because one or more of their fragments were lost is then 1-P^45. If we happen to have a packet loss probability of 0.01% (picked at random) then p = 0.0001, P is 0.9999, P^45 is 0.9955 and so the probability of any one message being lost is 0.45%. If the packet loss rate were 1% p is 0.01, P is 0.99, P^45 is 0.6362 and our message loss rate becomes ~36.38%.

[6] This very large value was made possible by tuning the values of net.core.rmem_max on the receiver to a very large value. It is 2x the 24M requested on the netperf command line because the Linux stack will tweak requested socket buffer sizes to account for packet buffer overheads.

Message Loss % versus Packet Loss %							
Packet Loss %	Fragments per Message						
	1	3	6	10	20	30	45
.001	.001	.003	.006	.01	.02	.03	.045
.003	.003	.009	.018	.03	.06	.09	.135
.01	.010	.030	.060	.100	.200	.300	.449
.03	.030	.090	.180	.300	.598	.896	1.341
0.1	0.100	0.300	0.599	0.996	1.981	2.957	4.402
0.3	0.300	0.897	1.787	2.960	5.832	8.619	12.646
1	1.000	2.970	5.852	9.562	18.209	26.030	36.381
3	3.000	8.733	16.703	26.258	45.621	59.899	74.606
10	10.000	27.100	46.856	65.132	87.842	95.761	99.127

This is why sending data which must be fragmented by IP is considered a bad idea. It is why TCP has mechanisms it uses to try to avoid IP fragmentation.

Let's run the UDP_STREAM test again, this time looking at the IP statistics having to do with fragment reassembly on the receiver:

```
receiver $ netstat -s
Ip:
     Forwarding: 2
     48577182 total packets received
     4 with invalid addresses
     0 forwarded
     0 incoming packets discarded
     1399126 incoming packets delivered
     301244 requests sent out
     6 outgoing packets dropped
     3 fragments dropped after timeout
     48226427 reassemblies required
     1048375 packets reassembled ok
     53 packet reassemblies failed

sender$ src/netperf -t UDP_STREAM -H 10.138.0.2 -- -R 1 -S 24M
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.2 () port
0 AF_INET : histogram : spin interval : demo
Socket  Message  Elapsed      Messages
Size    Size     Time         Okay Errors   Throughput
bytes   bytes    secs            #      #   10^6bits/sec
212992   65507   10.00      175248      0    9183.90
50331648           10.00      175245           9183.74

receiver $ netstat -s
Ip:
     Forwarding: 2
     56638701 total packets received
     4 with invalid addresses
     0 forwarded
     0 incoming packets discarded
     1574482 incoming packets delivered
     301329 requests sent out
```

```
6 outgoing packets dropped
3 fragments dropped after timeout
56287835 reassemblies required
1223620 packets reassembled ok
59 packet reassemblies failed
```

Again there were no socket buffer overflows (not shown). There were 3 messages lost as reported by netperf. The netstat statistics show 59-53 or 6 packet reassembles failed. This is greater than our lost message count. It is possible some datagrams got counted more than once for reassembly failure. If instead we look at reassembled OK statistics there were 175245 successful reassemblies. That happens to be equal to our number of successful receives.

There can sometimes be an OBOB (Off-By-One-Bug) somewhere[7]. In any event, it suggests quite strongly indeed that IPv4 datagram fragmentation was involved in our current losses.

Let us now run a test where we don't have any fragmentation. The instances here have an MTU of 1460 bytes, which means we should be able to use 1460 - 28 or 1432 bytes per message and not require fragmentation:

```
sender$ src/netperf -t UDP_STREAM -H 10.138.0.2 -- -R 1 -m 1432 -S 24M
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.2 () port
0 AF_INET : histogram : spin interval : demo
Socket  Message  Elapsed      Messages
Size    Size     Time         Okay Errors   Throughput
bytes   bytes    secs            #      #   10^6bits/sec
212992    1432   10.00     6410773      0    7344.12
50331648           10.00     6410773           7344.12
```

This time our sending rate was rather lower. Rather than a single send call with ~64KB of data in it, we are making send calls with 1432 bytes of data. Almost 46 calls where there used to be just 1. We go through UDP at both ends that many more times, and also make that many more receive calls at the receiver. Now, we will shift to:

[7] There can also be fragments which end up as orphaned and leave the reassembly queue only after a timeout. And depending on the speed of transmission, the 16-bit IPv4 datagram ID field can wrap quickly. This ID is used as part of the reassembly process. When there is packet loss, and the ID field wraps before that fragment reassembly timeout, you can get fragments from two different datagrams mistakenly put together. Such "Frankengrams" should then get dropped via mechanisms operating above the IP layer - for example via transport-layer checksums.

This can also result in losses in later test runs between two hosts if those runs are not separated by at least net.ipv4.ipfrag_time seconds. Some losses from test N can leave un-assembled datagram fragments in the reassembly area, and then the wrap of the IP ID space with test N+1 can cause those to be reassembled into Frankengrams caught by the transport-layer checksum. Odds are increased with larger values of net.ipv4.ipfrag_max_dist.

Pace Yourself!

As mentioned previously, UDP provides no flow control. Neither end-to-end, nor anything like TCP's congestion control to avoid overrunning parts of the network in between. That means that any "real" application using UDP in a non-trivial way must implement some sort of pacing. The netperf benchmark is no exception. Before building the netperf binary for these tests, the authors configured netperf via:

```
configure --enable-intervals --enable-spin --enable-burst
--enable-demo
```

Those first two are germane to this discussion. The first enables "intervals" mode in netperf, and the second enables spinning on a gettimeofday() to allow for much finer grained pacing than the interval timer used otherwise[8]. Unlike other benchmarks, netperf does not (currently) support setting an explicit bitrate via the command line. Instead, one specifies a number of sends to make in each interval (a global -b option), and the amount of time in each interval (a global -w option). It is left up to the benchmarker to pick values according to what she wishes to accomplish[9]. Let us assume we want to see ~1 Gbit/s. Let us also assume we will be sending 1024 bytes of data at a time and we want things to be as smooth as possible, so a "burst" of only one send at a time. We will be sending 8192 bits of data at a time. At 1 Gbit/s that would be 1000000000/8192 or 122070.3125 sends per second. If we invert that to get seconds per send that becomes 0.000008192 seconds per send or just a little bit more than 8 microseconds between sends. The finest netperf can handle at present is microseconds, so we'll use 8 microseconds:

```
sender$ src/netperf -t UDP_STREAM -H 10.138.0.2 -w 8u -b 1 -- -R 1 -m 1024 -S 24M
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.2 () port
0 AF_INET : histogram : spin interval : demo
Socket  Message  Elapsed      Messages
Size    Size     Time         Okay Errors   Throughput
bytes   bytes    secs            #      #   10^6bits/sec
212992    1024   10.00     1244932      0    1019.84
```

```
50331648           10.00     1244932           1019.84
```

Lo and behold! No packet losses! Is it repeatable? Let's check:

```
$ HDR="-P 1"; for i in `seq 1 10`
> do
> src/netperf $HDR -t UDP_STREAM -H 10.138.0.2 -w 8u -b 1 -- -R 1 -m 1024 -S 24M -O[10]
local_send_calls,remote_recv_calls
> HDR="-P 0"
> done

MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.2 () port
0 AF_INET : histogram : spin interval : demo
Local    Remote
Send     Recv
Calls    Calls

1245917  1245917
1247397  1247397
1243520  1243520
1244088  1244088
1244266  1244266
1247174  1247174
1235298  1235298
1243772  1243772
1244793  1244793
1247683  1247683
```

[8] This spinning means the netperf process will consume 100% of a CPU, no matter what its sending rate. The remote netserver process will consume only as much CPU as dictated by the rate.

[9] Per the authors' understanding, nuttcp and iperf3 instead have the benchmarker specify a bitrate. This is easier when one just wants a bitrate, but netperf's mechanism comes-in very handy in other situations where specifying by bitrate wouldn't apply. The nuttcp benchmark will behave like netperf with "spin interval" enabled - so it will consume 100% of the CPU on which it is running. The authors have not used iperf3's pacing, but know per Aaron Wood that prior versions of iperf3 had an issue with burstiness when pacing. Aaron has made fixes to the master branch for iperf3 that have made it into the 3.2 release.

[10] This is an "output selector" option selecting just those metrics of interest. In this case, the number of successful send calls at the sender and the number of successful receive calls at the receiver. A list of all the available output selectors can be seen by passing '?' as the argument to the -O option.

It would indeed seem that ~1 Gbit/s is sustainable in this current situation. Ten times in a row there were as many receives as there were sends. Of course, your mileage will no doubt vary.

Pick a Buffer Size, Any Buffer Size

Up until this point we have been using an enormous socket receive buffer size. The test-specific -S option has been used to set the remote socket receive buffer size. This has been possible because the maximum permitted value for an explicitly selected receive socket buffer size has been increased from the default of 212992 bytes to 48 MB (M = 1048576):

```
receiver $ sudo sysctl -a | grep rmem_max
net.core.rmem_max = 50331648
```

At this point a short digression into how socket buffer sizes work on Linux is in order. If an application creates a UDP datagram socket, initially the send and receive socket buffer sizes will be based on the following:

```
$ sudo sysctl -a | grep [rw]mem_default
net.core.rmem_default = 212992
net.core.wmem_default = 212992
```

We saw this earlier with our first UDP_STREAM test. If the application then makes an attempt to set a different socket buffer size for either send or receive, the Linux stack will first take the minimum of the passed-in value and the corresponding value among:

```
$ sudo sysctl -a | grep [rw]mem_max
net.core.rmem_max = 212992
net.core.wmem_max = 212992
```

And then take the maximum of twice that and a minimum size[11]. And it will silently set the socket buffer size to the result. That is, it will not return an error when the value it picked becomes different from the value passed-in.

To demonstrate, here are some results of a set of netperf tests asking for different sizes, or not asking and just accepting the defaults (when it was told "-1" on the command line). The

column titled “Result Tag” will show the value being passed-in (or not) to the setsockopt() calls:

```
$ HDR="-P 1";for i in -1 0 1 2303 2304 2305 4608 4609 65536 212991 212993 212994
1048576 `expr 1048576 \* 2`; do netperf $HDR -H localhost -l 1 -t UDP_STREAM -B $i --
-m 100 -s $i -S $i -O result_brand,rsr_size,lss_size; HDR="-P 0"; done
MIGRATED UDP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to localhost () port 0
AF_INET : demo
Result Remote      Local
Tag    Recv Socket Send Socket
        Size        Size
        Initial     Initial
"-1"   212992      212992
"0"    2304        4608
"1"    2304        4608
"2303"  4606        4608
"2304"  4608        4608
"2305"  4610        4610
"4608"  9216        9216
"4609"  9218        9218
"65536"  131072      131072
"212991"  425982      425982
"212993"  425984      425984
"212994"  425984      425984
"1048576"  425984      425984
"2097152"  425984      425984
```

This all means if you want an application to get a larger socket buffer size, and it makes explicit calls to set the socket buffer size, the corresponding _max sysctl parameter must be increased first. If the application makes no explicit calls to set the socket buffer size, the corresponding _default sysctl settings must be increased.

But Wait! There’s More!

While most folks will think in terms of the quantity of data in a packet, these limits in Linux are actually counting the bytes of buffer used to hold the packets. Consider a situation where a

[11] Defined via constants SOCK_MIN_RCVBUF and SOCK_MIN_SNDBUF, which would seem to be 2304 and 4608 bytes respectively.

NIC driver queues 1500-odd byte buffers to a NIC to hold incoming packets. Now consider an arriving stream of UDP datagrams carrying 256 bytes of data. Which of those numbers do you think is counted against the socket buffer's set size? Yep, the size of the buffer used to hold the packet data, not the packet data itself. So a receive socket buffer set to 1048476 bytes will not hold 1048576 / 256 or 4096 packets but 1048576 / 1500 or 699 packets.

It is the authors' opinion that it is best for applications to make explicit socket buffer size calls rather than rely on the default, as different applications have different needs. Not everyone shares that opinion.

If an application creates a TCP socket, and makes no explicit calls to set the socket buffer sizes, they will be sized based on:

```
$ sudo sysctl -a | grep tcp_[rw]mem
net.ipv4.tcp_rmem = 4096        131072   6291456
net.ipv4.tcp_wmem = 4096        16384   4194304
```

The first value of each of those three-tuples is the smallest value to which the socket buffer can shrink. The middle is the value at which it will be at the time the socket is created, and the third is the maximum value to which it will "autotune" at the hands of the Linux stack. Now we see why the values displayed for the TCP_STREAM test run at the beginning of this write-up were what they were.

If the application makes explicit calls to set the socket buffer size for the TCP socket, the behavior and limits of those calls will be the same as for explicit calls against a UDP socket as described above.

Ok, so we've finished the "brief" digression into how socket buffer size settings behave under Linux. Let's get back to picking a "good" value for a socket buffer size for a UDP application. In very simple terms, the flow of a UDP datagram into an instance looks like:

1. Packet arrives at the host holding the instance
2. Packet goes through some "plumbing" in the host
3. Packet is queued to the instance's vNIC's receive queue
4. Instance is told there is a packet
5. Instance pulls the packet off the vNIC's receive queue
6. Packet goes up the stack in the instance
7. Data from the packet is queued in the socket receive buffer for the application
8. Application is told there is a packet
9. Application reads the data from the socket receive buffer and does something with it

The socket receive buffer acts as, well, a buffer between the stack in the instance and the application. Messages arriving while the application is busy will be stored in the socket receive buffer to wait until the application can get to them. If there is a large burst, or if the application is held-off from running for "long enough" while messages keep arriving then arriving messages will fill the buffer and some will be dropped.

If the application is just a simple, one-at-a-time request/response application, or perhaps it is known there are "never" more than N requests to the application in flight at one time before requests stop coming while responses are awaited, then one can set the socket buffer size based on N and the maximum message size. In that case, no matter how long the receiving application is held-off from taking messages out of the socket buffer, there should be no socket buffer overflows.

Similarly, if the application is a bulk transfer, and it happens to implement a roughly TCP-like flow control mechanism with a receive window and ACKnowledgements and such then the socket buffers can be set based on that window. Again, so long as there are no bugs in the application's flow control, there should be no socket buffer overflows.

But not all applications are like that. They just send data, hopefully at a steady pace, but still without waiting for acknowledgements. Video streaming can be like that. Certainly a netperf UDP_STREAM test will be like that. What we need to "know" then is how long the application can go between pulling messages from the socket buffer, at what rate those messages might be arriving, and whether or not some might end-up arriving in a burst.

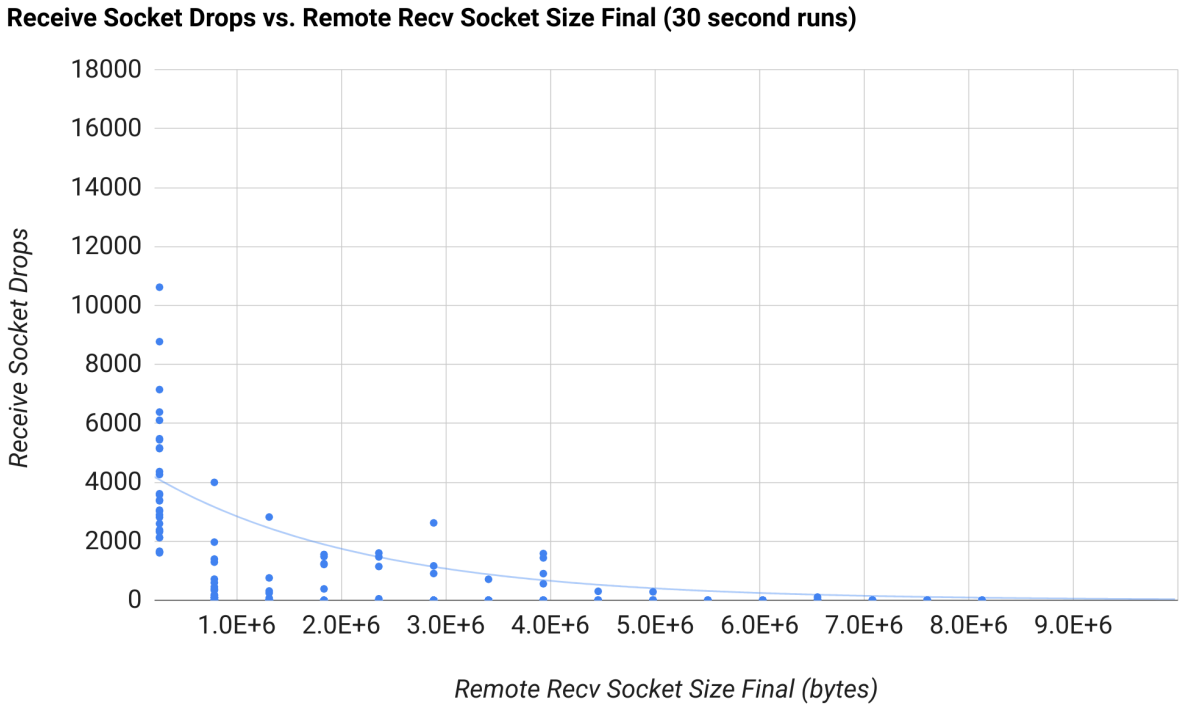
Let's go back to our 1024 byte messages case. If we ignore any bursts, assume that when the receiving application consumes data it does so at infinite speed, and our 1024 byte message consumes only 1024 bytes of the receive socket buffer without any per-message buffer overhead[12] we can see the following theoretical times to fill a given socket buffer with data arriving at various bit rates. This then is the maximum length of time the receiving application could be held-off before socket receive buffer drops started happening:

[12] The Linux networking stack puts packets into packet buffers which can be larger than the packets themselves. When it queues data to a socket, it counts the overhead of the actual buffer size against the socket buffer limit. This is the origin of that doubling of the request value when setting socket buffer sizes.

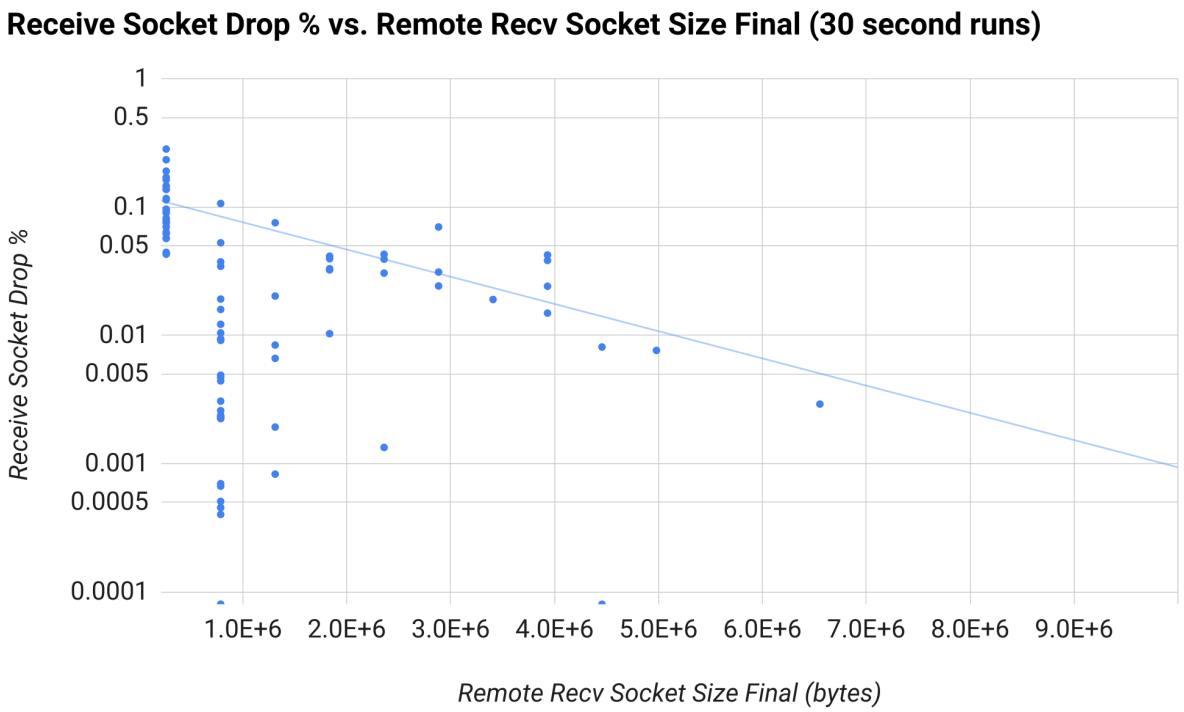
Time To Fill Socket Buffer (milliseconds)							
Socket Buffer Size (bytes)	Arrival Rate (Mbit/s)						
	10	30	100	300	1000	3000	10000
262144	209.715	69.905	20.972	6.991	2.097	0.699	0.210
786432	629.146	209.715	62.915	20.972	6.291	2.097	0.629
1310720	1048.576	349.525	104.858	34.953	10.486	3.495	1.049
1835008	1468.006	489.335	146.801	48.934	14.680	4.893	1.468
2359296	1887.437	629.146	188.744	62.915	18.874	6.291	1.887
2883584	2306.867	768.956	230.687	76.896	23.069	7.690	2.307
3407872	2726.298	908.766	272.630	90.877	27.263	9.088	2.726
3932160	3145.728	1048.576	314.573	104.858	31.457	10.486	3.146
4456448	3565.158	1188.386	356.516	118.839	35.652	11.884	3.565
4980736	3984.589	1328.196	398.459	132.820	39.846	13.282	3.985
5505024	4404.019	1468.006	440.402	146.801	44.040	14.680	4.404
6029312	4823.450	1607.817	482.345	160.782	48.234	16.078	4.823
6553600	5242.880	1747.627	524.288	174.763	52.429	17.476	5.243
7077888	5662.310	1887.437	566.231	188.744	56.623	18.874	5.662
7602176	6081.741	2027.247	608.174	202.725	60.817	20.272	6.082
8126464	6501.171	2167.057	650.117	216.706	65.012	21.671	6.501

What you see next is a chart of receive socket buffer drops versus the size of the receive socket buffer, for repeated runs of a netperf UDP_STREAM test transferring 1024 byte messages at roughly 1 Gbit/s.

The first receive socket buffer size was 262144 bytes, and it increases by 524288 bytes each time. The y-axis is the total number of socket buffer overflows over the duration of a given test.



The solid line is an exponential trend fit to the data points by the spreadsheet application. There were thirty runs at any one receive socket buffer size, measured in groups of ten across three days. You can see that once the receive socket buffer size was greater than 4 million bytes, receive socket buffer overflows became quite rare. The little, blue half dots you see along the x-axis are the runs with zero receive socket buffer drops stacked one on top of another. By the time the socket buffer was around 8 million bytes they were nearly non-existent. Another way to look at the results is the number of receive socket buffer drops as a percentage of the number of messages sent:



Runs without drops are not visible here because their percentages were 0 and the y-axis is the spreadsheet application's presentation of a logarithmic scale. There were only a few runs with receive socket buffer drops once the receive socket buffer was larger than four million bytes. There were none once the receive socket buffer was larger than about six and a half million bytes. So, we can presume for our given test a socket buffer of 8MB is sufficient to account for times the netserver was held-off from reading from the socket, and/or to absorb any "bursts" of arriving traffic, if say the receiving instance itself was precluded from draining the vNIC receive queue. At least in this case where the nominal transfer rate is 1 Gbit/s. If the nominal transfer rate were higher, we would almost certainly need to have a larger buffer.

With these instances, the vNIC receive queue happens to have been 4096 entries. The relationship between the instance and the host is similar to that between the receiving application and the networking stack within the instance, and the vNIC receive queue performs a function similar to that of the receive socket buffer.

Sometimes There Just Aren't Enough Socket Buffers

Even with a Very Large (™) receive socket buffer, there can be situations where there are still receive socket buffer overflows. Any sustained situation with the sender faster than the receiver will result in overflows - at least unless perhaps one configures an Enormously Large (™) receive socket buffer large enough to hold all the traffic of the test. And that method

becomes somewhat difficult with longer tests. So.... Consider the following test with a 64 VCPU instance running Ubuntu 20.04 (Linux 5.15 kernel) sending to a 16 VCPU instance running the same bits[13]:

```
OMNI Receive TEST from lg1.c.mumble.internal () port 0 AF_INET to sut.c.mumble.internal () port 0 AF_INET : histogram : spin interval
: demo
Elapsed Remote Remote      Local       Remote  Local  Remote     Local      Local   Local   Remote  Remote  Result
Time    Send   Send Socket Recv Socket Send    Recv   Send       Recv       Peak    Peak    Peak    Peak    Tag
(sec)   Size   Size        Size        Calls   Calls  Throughput Throughput Per CPU Per CPU Per CPU Per CPU
               Final       Final                                            Util %  ID      Util %  ID
10.00   1      212992      268435456   6094436  6094436  4.88       4.88       44.56   7       99.60   63              0
10.00   16     212992      268435456   6562937  6562816  84.00      84.00      42.74   7       87.81   63              0
10.00   128    212992      268435456   6019098  6019046  616.35     616.34     45.86   7       99.70   63              0
10.00   1024   212992      268435456   8054958  8054823  6598.55    6598.44    72.18   13      61.46   9               0
10.00   1432   212992      268435456   7371640  6895164  8444.81    7898.96    71.14   7       51.74   1               476390
10.00   2048   212992      268435456   4003726  3923808  6559.61    6428.68    83.67   1       42.76   32              0
10.00   4096   212992      268435456   3551881  3551708  11638.65   11638.08   93.92   1       72.37   10              0
10.00   8192   212992      268435456   1476532  1476508  9676.46    9676.30    70.76   1       90.56   32              0
10.00   8972   212992      268435456   1469570  1469541  10547.85   10547.64   79.53   1       55.76   0               0
```

Apologies for the miniscule font - the perils of trying to put wide, fixed-width output into a narrow document. What you see here are a series of tests, going from 1 byte to 8972 bytes per UDP message. The send socket buffer size is default, the receive is set to Very Large (™). You can see the number of send and receive calls, the send and receive throughput in Mbits/s, and CPU utilization information. At the tail end, where anything specified as the result tag would be, we have the results of some kludgery to get socket buffer overflows into the output[14]. The netperf command used a test-specific -P option to always use the same four-tuple. Going by the ID of the peak utilized CPU in the receiver, we can see that before IP fragmentation (1432 bytes and below, as GCP instances have an MTU of 1460 bytes) all the traffic (with one exception) was handled on CPU 7. Once it was fragmented, that switched to CPU 16. What happens then if we bind netperf (by adding global flag "-T $netperf_cpu,") to a CPU other than the one taking inbound interrupts? Say CPU 4:

```
OMNI Receive TEST from lg1.c.mumble.internal () port 0 AF_INET to sut.c.mumble.internal () port 0 AF_INET : histogram : spin interval
: demo
Elapsed Remote Remote      Local       Remote  Local  Remote     Local      Local   Local   Remote  Remote  Result
Time    Send   Send Socket Recv Socket Send    Recv   Send       Recv       Peak    Peak    Peak    Peak    Tag
(sec)   Size   Size        Size        Calls   Calls  Throughput Throughput Per CPU Per CPU Per CPU Per CPU
               Final       Final                                            Util %  ID      Util %  ID
10.00   1      212992      268435456   6024940  6024927  4.82       4.82       47.47   7       99.90   63              0
10.00   16     212992      268435456   6013921  6013878  76.98      76.98      46.61   7       99.90   63              0
10.00   128    212992      268435456   6011182  6011136  615.54     615.53     48.55   7       99.90   63              0
10.00   1024   212992      268435456   8293744  8293663  6794.16    6794.10    73.00   4       50.00   9               0
10.00   1432   212992      268435456   6075479  6075406  6959.99    6959.91    57.86   4       47.25   32              0
10.00   2048   212992      268435456   3729274  3692387  6109.98    6049.54    78.25   1       40.82   31              0
10.00   4096   212992      268435456   3008697  3008623  9858.80    9858.55    81.60   1       51.05   10              0
10.00   8192   212992      268435456   1826618  1826584  11970.79   11970.57   84.00   1       78.00   10              0
```

[13] See the section on netperf-fu for an explanation of what was done to generate results looking like that.

```
10.00   8972   212992     268435456   1425204  1425175  10229.43   10229.22   76.65   1      50.95   32           0
```

We can see that with a couple exceptions, until IP fragmentation, the highest CPU utilization is on the CPU to which netperf ("Local") was pinned and that it wanted well more than ½ a CPU. And we can see that the socket buffer overflows have gone away. This means the issue of receiving application not being as fast as the sending application[15] has been eliminated as a source of packet loss - this time at least. It still remains theoretically possible without some form of explicit flow control. We can also see that IP fragment reassembly can be rather expensive. CPU 16, the one taking the interrupts for fragmented traffic, runs often at ~80-100% utilization. Expensive enough that in the first set of tests, it was probably inducing the process scheduler to schedule the application elsewhere. It may have also meant that fragment reassembly wasn't able to keep-up with the inbound rate[16].

Conclusion

There are several considerations when benchmarking UDP bulk flows. In particular, one must take into account socket buffer sizes. Under Linux the default and maximum for UDP sockets are too low by default for anything but the lowest transfer rates. Even with larger socket buffer sizes, applications and benchmarks sending bulk data over UDP need to provide some form of flow control. Receive socket buffer overflows are not the only source of packet loss for UDP bulk flows. Finally, applications/benchmarks must be conscious of the effects of IP datagram fragmentation when selecting message sizes because IP fragmentation amplifies a packet loss rate on the network into an even higher message loss rate. IP fragment reassembly can also be expensive in CPU cycles.

Netperf-fu

A few tidbits on using netperf. The initial runs used for this writeup used more "classic" netperf command lines - for example a UDP_STREAM test with a 1 KiB message size and a 24 MiB (requested) receive socket buffer:

```
netperf -t UDP_STREAM -H 10.138.0.2 -w 8u -b 1 -- -R 1 -m 1024 -S 24M
```

Netperf does not know how to retrieve socket buffer overflow stats, and there is no UDP_INFO getsockopt() so the socket buffer overflows have to be looked-for out-of-band from the command above. With a bit of script kludging however, that can be changed.

---

[14] This was to always take the same path through the receiving instance's virtio queues and so vCPUs.

[15] Strictly speaking, the receiving application being as fast as traffic enters the instance.

The “omni” tests of netperf include the ability to have a “UDP_MAERTS” like test - MAERTS being STREAM spelled backwards - with the direction of data flow going from netserver to netperf. While one cannot just say “UDP_MAERTS” one can run:

```
netperf -H $sut_ip -t omni  -- -s 128M -m ,1024  -T udp -d recv -L $lg1_ip -R 1
```

on a system called lg1 and netserver on the system called sut will send 1024 byte UDP messages to netperf on lg1. The test-specific -L option is definitely required or netserver will not know where to send messages. One can include “omni output selectors” with the -O (or -o) option to specify what is reported, and if one has “result_brand” as the last one, but without a global -B option to set a result brand (tag) there will be room at the end of an output line for something else. For example, the number of socket buffer overflows reported via netstat -s over the test:

```
BEFORE=`netstat -s | grep "receive buffer errors" | awk '{print $1}'`
RES=`netperf -H $sut_ip -t omni -c -C -- -s 128M -m ,128 -T udp -d recv -L $lg1_ip -R
1 -P 65432 -O
remote_send_size,remote_send_throughput,local_recv_throughput,local_cpu_peak_util,loc
al_cpu_peak_id,remote_cpu_peak_util,remote_cpu_peak_id,result_brand | grep -v [a-z]`
AFTER=`netstat -s | grep "receive buffer errors" | awk '{print $1}'`
echo "$RES" `expr $AFTER - $BEFORE`
128    553.53     541.44     89.10   25      100.00  9               98899
```

The grep in the netperf command pipeline is to get just the results. Of course, that isn’t overly useful unless one memorized the order of the output selectors. One can run a “sacrificial” netperf test to get the headers - just run the same command, but instead of piping it to that grep, pipe it to head -5. So, the command sequence used elsewhere when discussing pinning the receiver ends-up looking like:

```
netperf -H $sut_ip -t omni -c -C -- -s 128M -m 1 -M 1 -T udp -d recv -L $lg1_ip -R 1
-O
elapsed_time,remote_send_size,rss_size_end,lsr_size_end,remote_send_calls,local_recv_
calls,remote_send_throughput,local_recv_throughput,local_cpu_peak_util,local_cpu_peak
_id,remote_cpu_peak_util,remote_cpu_peak_id,result_brand | head -5;\
```

[16] Eric Dumazet has submitted patches to upstream Linux which greatly improve the performance of IP fragment reassembly. They are, we believe, in the version 4.17 or later kernels.

```
for m in 1 16 128 1024 1432 2048 4096 8192 8972; \
do \
BEFORE=`netstat -s | grep "receive buffer errors" | awk '{print $1}'` ;  RES=`netperf
-H $sut_ip -t omni -c -C -- -s 128M -m ,${m} -M ,${m} -T udp -d recv -L $lg1_ip -R 1
-P 65432 -O
elapsed_time,remote_send_size,rss_size_end,lsr_size_end,remote_send_calls,local_recv_
calls,remote_send_throughput,local_recv_throughput,local_cpu_peak_util,local_cpu_peak
_id,remote_cpu_peak_util,remote_cpu_peak_id,result_brand | grep -v [a-z]` ;  \
AFTER=`netstat -s | grep "receive buffer errors" | awk '{print $1}'`;  \
echo "$RES" `expr $AFTER - $BEFORE`; \
Done
```

And we have the (imperfect) illusion of netperf reporting UDP socket buffer overflows.

Acknowledgements

We would like to thank our colleague Aaron Wood for his iperf3 feedback.