Google Cloud

Considerations When Benchmarking TCP Bulk Flows

Some Tunes Dedicated to Your Interconnect

Rick Jones, Derek Phanekham

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

The Linux networking stack is generally a well-tuned machine, but one size does not always fit. This write-up will attempt to describe some of the situations where it will be necessary to tweak the stack's settings. This is particularly true when looking to transfer data at high speed across a non-trivial distance. Some of the settings discussed and recommendations given here are specific to Google Cloud environments, but much of it is more broadly applicable to general Linux networking.

Please note, the performance figures presented in this document are merely examples of what was achieved and serve to illustrate the concepts being discussed. They do not represent any guarantee of achievability.

TL;DR: Just Tell Me What to Look for and Change

You have just run a benchmark and gotten lower-than-expected throughput. These steps will likely cover the common cases. However, TCP performance is a maze of twisty passages, all interconnecting. Details matter and a TL;DR cannot cover all of them.

1. Is the sending instance "large enough" to have an egress (i.e. outbound) network cap large enough to be allowed to send data at the expected speed?
   As of this update, instances (VMs) in Google Cloud are limited to different egress bandwidth limits depending on the VM family, the number of vCPUs, Internal versus External IP address, and whether Tier_1 networking is enabled.[1] If the sending instance is too small to send at the desired rate, an instance with more vCPUs should be allocated. Or, if the instance is of a compatible type and high-enough vCPU count, enable Tier_1 networking for the VM.
   There is no ingress (i.e. inbound) cap for an Internal IP placed on an instance by Google Cloud. This of course may change. Additional documentation on network caps can be found here and here.
2. What is the round-trip-time (RTT) between the sender and receiver in seconds? And if the benchmark or application is *not* making explicit calls to setsockopt() (see below), what is the third value for net.ipv4.tcp_rmem on the receiver and net.ipv4.tcp_wmem on the sender?[2] Both of those should be large enough such that:

[1] This is shared with writes to non-local PD storage. (https://cloud.google.com/compute/docs/disks#performance)
[2] Or their equivalents for OSes other than Linux.

$ReceiversTcpRmem >= TotalDesiredBitsPerSecond / 4 * RTT / StreamsToAchieve$
and
$SendersTcpWmem >= TotalDesiredBitsPerSecond / 8 * RTT / StreamsToAchieve$[3]

If the system at one end or both is small, or there will be many of these connections in parallel, **`net.ipv4.tcp_mem`** may need to be tweaked as well. That setting controls how much memory will be used by TCP across all connections, in units of pages rather than bytes.

3. If the benchmark or application *is* making explicit setsockopt() calls (see below), then repeat the previous bullet item but with net.core.rmem_max on the receiver and net.core.wmem_max on the sender.
4. Does the application itself have a limit to how much data it is willing to have outstanding at one time before waiting for an application-level response? If so, is that large enough to allow the receiver's advertised window to be filled? One common case, which we will look at later in the document, is scp/ssh transfers.
5. If netstat statistics on either end (netstat -s) show non-trivial numbers of reorders and Duplicate SACKS (aka DSACKS) consider increasing the values for:
   net.ipv4.tcp_reordering
   net.ipv4.tcp_max_reordering
   To make TCP more resilient in the face of the reordering happening in the network.
6. If the RTT between sender and receiver is particularly high it can take longer for a TCP connection to "get up to speed." Most benchmarks appear to default to 10 seconds of runtime and this can be insufficient.
7. If the kernel on the sender is based on version 4.20 or later, switch to "BBR" congestion control instead of Cubic. BBR congestion control is more resilient in the face of occasional packet losses and maintains a larger congestion window.

   - `sudo sysctl -w net.ipv4.tcp_congestion_control=bbr`

8. If BBR cannot be used, and the RTTs are high, consider disabling some or all of Cubic's "hystart_detect" heuristics:

   ```
   echo 0 > /sys/module/tcp_cubic/parameters/hystart_detect
   ```
   or
   ```
   echo 2 > /sys/module/tcp_cubic/parameters/hystart_detect
   ```

9. If there are lengths of time when one of the CPUs of the instance(s) saturates, corresponding to a drop in the transfer rate reported for an interval and the Linux

[3] People will often tune send and receive to the same value. That is fine, so long as receive is large enough. It is generally "OK" for send to be larger than necessary.

kernel is based on version 4.4 through 4.14 (inclusive) consider upgrading to a kernel based on 4.15 or later. With Rather Large Windows (™) and scattered packet losses, some inefficient behavior in TCP marking packets lost or not can result in long periods of CPU saturation. This behavior is fixed in the version 4.15 and later kernels.

10. Does the other system have the proven ability to transfer data at the desired rate? For example, if testing between Google Cloud and on-premises, if the Google Cloud instance is large enough to achieve the desired rate based on its egress throttle, does the on-premises system have the ability to achieve that rate as well?
11. Are you able to increase the MTU that you are using to send data? If the data is between machines on a Google Cloud VPC, it is possible to increase the MTU to 8896 bytes when creating the VPC. This will allow machines to send with larger packet sizes, which can yield higher and more stable throughput. However, if you are sending traffic across a link that only supports a lower MTU, traffic will be sent at the lower MTU. Some additional considerations must be taken into account if using a service that further encapsulates your packets, such as Cloud VPN.

Linux, and sysctl and setsockopt(), Oh My!

Before we go further, we should discuss the interaction in Linux between sysctl settings and setsockopt() calls to set socket buffer sizes. And which sysctl settings come into play when an application[4] makes such setsockopt() calls, and which when the application does not.

If the application makes an explicit setsockopt() call to set a socket buffer size for either send (SO_SNDBUF) or receive (SO_RCVBUF), the Linux stack will first take the minimum of the passed-in value and the values for:

```
$ 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[5]. And it will *silently* set the socket buffer size to the result. That bears repeating - it will ***silently*** set the socket buffer size to the result.

If the application does not make any attempts to set socket buffer sizes, then for TCP they will be dynamically sized based on:

[4] In this context "application" means anything running above the "socket layer" - be that library code, middleware, or actual application code.
[5] Defined via constants SOCK_MIN_RCVBUF and SOCK_MIN_SNDBUF, which would seem to be 2304 and 4608 bytes respectively.

```
$ sudo sysctl -a | grep tcp_[rw]mem
net.ipv4.tcp_rmem = 4096        87380   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 be shrunk if Linux deems it necessary. 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.

Here are some results of a series of netperf commands where it either does not make setsockopt() calls, or calls setsockopt() with a particular value. The system had the sysctl settings above. A requested socket buffer size of "-1" tells netperf to make no setsockopt() call.

```
sender@instance-1:~$ HDR="-P 1";for i in -1 0 2304 2305 4608 4609 65536 212991
212993 212994 1048576 `expr 1048576 \* 2`; do netperf $HDR -H localhost -l -100 --
-m 100 -s $i -S $i -O rsr_size_req,rsr_size,lss_size_req,lss_size; HDR="-P 0"; done
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to localhost () port
0 AF_INET : histogram : demo
Remote      Remote      Local       Local
Recv Socket Recv Socket Send Socket Send Socket
Size        Size        Size        Size
Requested   Initial     Requested   Initial
-1          131072      -1          16384
0           2304        0           4608
2304        4608        2304        4608
2305        4610        2305        4610
4608        9216        4608        9216
4609        9218        4609        9218
65536       131072      65536       131072
212991      425982      212991      425982
212993      425984      212993      425984
212994      425984      212994      425984
1048576     425984      1048576     425984
2097152     425984      2097152     425984
```

So, you can see that when netperf didn't make any setsockopt() calls, it got socket buffer sizes based on net.ipv4.tcp_[rw]mem, and when it did make setsockopt() calls, it got socket buffer sizes based on net.core.[rw]mem_max.

Sending Instance Size

In broad terms, n2 instances in Google Cloud have their outbound (i.e. egress) throughput capped to 2 Gbits/s multiplied by the number of vCPUs with a minimum of 10 Gbit/s[6] and a maximum of 32 Gbit/s. So, a single-vCPU instance will be limited to 2 Gbit/s; that will increase to 10 Gbits/s for instances with 2 through and including 5 vCPUs and by an additional 2 Gbit/s for each additional vCPU from 6 through 16 vCPUs. After that, additional vCPUs will not increase the outbound throughput cap. However, if the VM is of a supported family and has a sufficient vCPU count, one may enable Advanced/Tier_1 networking for the VM. For additional details concerning instance network bandwidth, please refer to the public documentation.

This cap is shared with storage writes. As of this writing, there is no inbound (ingress) throughput cap applied to instances for Private IP traffic. That however is subject to change.

Keep in mind, this is not a per-vCPU or per-vNIC limit, but a per-instance limit. Keep in mind as well, this is "guaranteed not to exceed" rather than a "guaranteed to achieve."

For all of the tests in this article, we will be using n2-standard-8 instances.

TCP "Speed-of-Light"

This outbound throughput cap is for the "total" bitrate sent by the instance and includes protocol headers. Most if not all benchmarks (eg netperf, iperf3, etc) and applications will report throughput based on just the bits of data exchanged without the protocol headers. Thus one should not expect a given benchmark to report the *full* bitrate because some of that bitrate is consumed by the protocol headers. Generally speaking, the "to/from user space" limit will be:

$$(Data\ /\ (Data\ +\ Headers))\ *\ Linerate$$

Most of the time in a bulk transfer $Data$ will be the Maximum Segment Size (MSS) for the connection. For example, this test between two n2-standard-8 ($Egress\ cap/linerate\ =\ 16\ Gbit/s$) instances in the same zone with the default VPC MTU of 1460 bytes[7]:

[6] For instances with 2 vCPUs or more. A single-vCPU instance such as n2-standard-1 has a cap of 2 Gbit/s.
[7] us-east4-b in this case

```
senderm@instance-1 netperf -H instance-2 -- -O
rsr_size_end,lss_size_end,send_size,elapsed_time,throughput,transport_mss
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.11 ()
port 0 AF_INET : histogram : demo
Remote      Local        Elapsed Throughput Transport
Recv Socket Send Socket  Time               MSS
Size        Size         (sec)              bytes
Final       Final
6291456     4194304      10.00   15602.15   1408
```

Achieved 15602 Mbit/s. For each TCP segment sent there would have been a TCP header of 32 bytes[8] and IPv4 header of 20 bytes and an Ethernet header of 14 bytes. Each TCP segment carries 1408 bytes, so we should not expect to see more than

$$1408/(1408 + 32 + 20 + 14) * Linerate$$

or

$$0.955 * Linerate$$

Those of you quick with your slide rules will notice that the test actually achieved 0.975 of LineRate. Why? Because the limit is based on what is pulled from the instance rather than pushed to "the wire"[9] and Google Cloud supports TCP Segmentation Offload (TSO) in the gVNIC driver. This means for a bulk transfer the instance is (usually) sending larger-than-MSS (Maximum Segment Size) "uber segments'' to the plumbing with just the one TCP header which the plumbing will turn into N "typical" TCP segments on the wire. That header gets used as a template for the on-the-wire TCP headers. Thus it appears we have exceeded the TCP "speed-of-light" for this situation but we haven't really. All in all, if one achieves 85% to 90% of "linerate" one is doing quite well.

[8] This includes the TCP Timestamp option, which one should leave/have enabled.

[9] And 99 times out of 10 (sic) "the wires" over which the traffic travels have bitrates >> 16 Gbit/s. The 100th time out of ten can be when there is a physical link in the network that has 10 Gbit/s links in some places. For example over a 10 Gbit/s Interconnect. This is one reason why one will not always see a single-stream at 15 Gbit/s even between sufficiently large endpoints/instances. There can also be situations where a VM has an egress cap as large as the physical bitrate available to the host on which it runs. And when that happens, the original equation will prevail and then-some, even with aggregate streams, because there is an additional encapsulation header for each segment. This will be the case for "whole host" slice-of-machine VMs with Tier_1 networking enabled.

Round-Trip-Time and Window Sizes

The chief limit of TCP bulk-transfer performance is receive window... receive window and send socket buffer. The two limits to TCP performance are receive window and send socket buffer ... and congestion window. The three limits to TCP performance are receive window, send socket buffer and congestion window ... and application window. The four ... err, no .. Amongst the limits to TCP performance are receive window, send socket buffer size, congestion window and application window. Why are you standing there facing us with a large trout in your hand?[10]

Receiver Side TCP

Unlike UDP, TCP includes end-to-end flow control. This is to prevent a fast sender from overwhelming a slower receiver. Broadly speaking it works by having a receiving TCP advertise a "window" into which a sending TCP may send data. This is expressed as a number of data bytes. The sending TCP may send no more than "window" data bytes at one time before it must stop and wait for a window update to arrive from the receiving TCP.

As the receiving application pulls data from the connection, the receiving TCP will send window updates to the sending TCP enabling the sending TCP to send more data. This happens essentially continuously and is referred to as a "sliding window." Thus, when things are well-tuned and working well, there is a continuous stream of data in one direction and window updates in the other.

Since a sending TCP will not receive a window update from the receiver sooner than one round-trip-time, we can express one of the fundamental limits to TCP transfer performance:

$$Throughput <= WindowSize / RoundTripTime$$

We can rearrange the terms to arrive at a formula for computing the needed window size:

$$WindowSize >= Throughput * RoundTripTime$$

This is where the setting for **`net.ipv4.tcp_rmem`** comes into play when benchmarks/applications are not making explicit calls to setsockopt() to set socket buffer sizes. The third value of that sysctl controls

[10] https://en.wikipedia.org/wiki/The_Fish-Slapping_Dance and https://en.wikipedia.org/wiki/The_Spanish_Inquisition_(Monty_Python)

the limit to which Linux will allow the receive socket buffer (aka SO_RCVBUF) to grow. As the Linux networking stack sees fit it will grow the receive socket buffer limit. As that limit increases, Linux TCP will advertise a larger window to the sender.

It isn't actually a one-to-one relationship. The Linux networking stack counts not just the actual bytes of data queued to a socket. It also counts the total bytes of buffering used to hold that data. And tcp_rmem really specifies the limit for the latter. So, if 1024 bytes of data arrive in a TCP segment, and that is in the stack in a 2048 byte packet buffer, 2048 is what will be counted against tcp_rmem.

Linux TCP doesn't really know in advance how efficiently arriving packets will be buffered. It is also extremely inefficient to always copy data around and Linux strives to avoid that. So for a given **`SO_RCVBUF`** size what TCP window should be advertised? If the full SO_RCVBUF were advertised, data could arrive as a series of 1 byte segments in largish buffers and that would mean either going way past tcp_rmem in memory consumed, or data copies. Data might arrive as large segments with almost no wasted space in the buffers, but the stack cannot count on that. Linux splits the difference here and assumes that on average buffers will be at least half-full and advertises a TCP window of ½ **`SO_RCVBUF`**.

If we are willing to have multiple streams, each does not have to have a window large enough to achieve the desired bitrate on its own.

This is why we arrive at:

$$ReceiversTcpRmem \geq \frac{TotalDesiredBitsPerSecond}{4} * \frac{RTT}{StreamsToAchieve}$$

The units of **`net.ipv4.tcp_rmem`** are bytes. Linux will advertise ½ of that as the window, so we would multiply by two. At the same time, we want to convert from bits to bytes, so we would divide by eight. Thus we end-up dividing by 4. We divide by StreamsToAchieve - how many streams we are willing to use to achieve TotalDesiredBitsPerSecond.

Sending Side TCP

A sending TCP must retain a reference to transmitted data until that data has been ACKnowledged by the receiving TCP. It does this in case the data was lost along the way and has to be re-sent. How much data a sending TCP will retain is determined by the limit of the send socket buffer (SO_SNDBUF) and that is controlled (again, in the case where the

benchmark/application does not make an explicit setsockopt(SO_SNDBUF) call) via a sysctl setting:

```
net.ipv4.tcp_wmem
```

As data is ACKnowledged it frees-up space against this limit, and the sending TCP can send more data, assuming the receiver has advertised enough window. So, even if a receiving TCP advertised an infinite window, the sending TCP would not send any more data than it could track before having to stop and wait for an ACKnowledgement. Just as with a window update the soonest this can happen is one round-trip-time, which means we can re-use:

$$Throughput \leq WindowSize \; / \; RoundTripTime$$

only with the tcp_wmem setting. In this case, the Linux TCP code has control over the buffering and it does not have to make an assumption about buffer efficiency. So, we can compute the tcp_wmem value required for a given bitrate without the factor of two:

$$SendersTcpWmem \geq \frac{TotalDesiredBitsPerSecond}{8} * \frac{RTT}{StreamsToAchieve}$$

In the name of simplicity, we could simply re-use the computation for tcp_rmem - it would still "work" but with the chance that we would have more data queued in the sending TCP than we really needed.

TCP Congestion Window

So far, we have discussed the receiving TCP's advertised window, and how much data the sending TCP can track at one time. There is a third window to consider, the congestion window. This is where considerations of congestion control heuristics and packet losses come into play. All the subtle nuances of the different TCP congestion control algorithms are beyond the scope of this writeup, and the authors would no doubt botch them anyway, so we will consider just two, and not get deep into their details - the default congestion control in Linux known as Cubic, and a second one introduced with the Linux 4.9 kernel called BBR.

An excellent resource for BBR can be found in a paper written by some of its creators.

The sending instance in these examples is running a 6.2.0 kernel.

We will use an extreme example here. We have two Google Cloud instances located in Amsterdam (europe-west4) and Oregon (us-west1) respectively. As of this writing, the round-trip time between them using private IPs is on the order of 144 milliseconds:

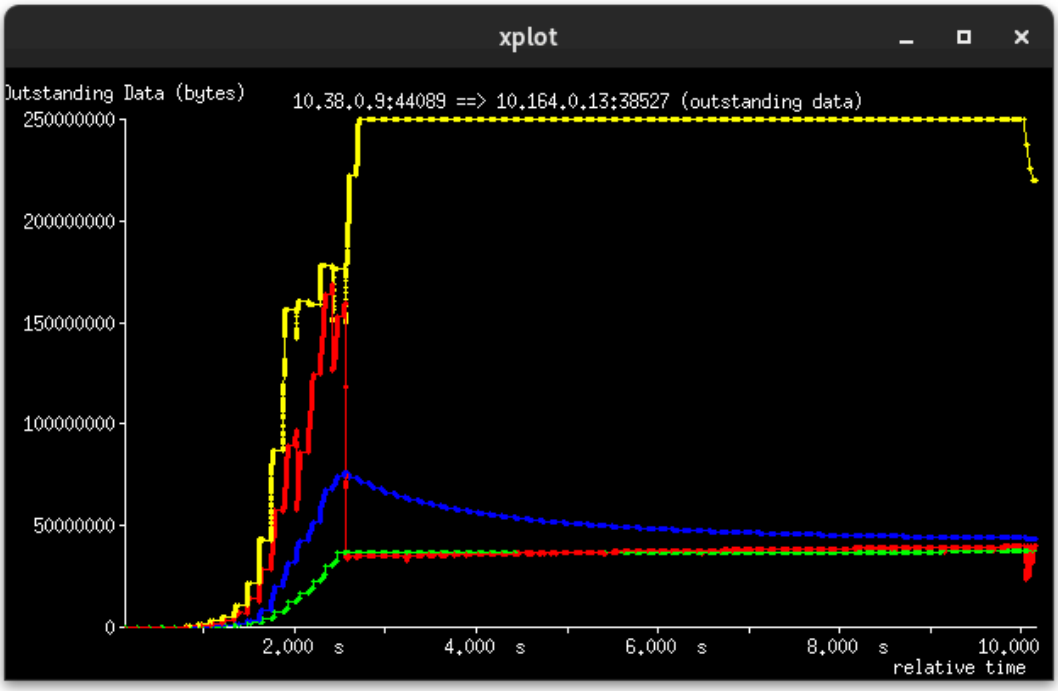
```
sender@amsterdam:~$ ping -c 100 -q oregon-2204
PING 10.138.0.12 (10.138.0.12) 56(84) bytes of data.
--- 10.138.0.12 ping statistics ---
100 packets transmitted, 100 received, 0% packet loss, time 99042ms
rtt min/avg/max/mdev = 144.414/143.458/145.650/0.121 ms
```

So, what does a ten second test sending data from Oregon to Amsterdam look like when we've already tweaked tcp_rmem and tcp_wmem to more than compensate for the Bandwidth Delay product? We will use Cubic as the congestion control heuristic:

```
receiver@amsterdam:~$ netperf -H oregon-2204 -t TCP_MAERTS -D -1.0
MIGRATED TCP MAERTS TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.9 () port
0 AF_INET : histogram : demo
Interim result:    6.36 10^6bits/s over 1.131 seconds ending at 1706992893.671
Interim result:  745.49 10^6bits/s over 1.206 seconds ending at 1706992894.877
Interim result: 3785.60 10^6bits/s over 1.014 seconds ending at 1706992895.891
Interim result: 3235.59 10^6bits/s over 1.046 seconds ending at 1706992896.937
Interim result: 3544.52 10^6bits/s over 1.000 seconds ending at 1706992897.937
Interim result: 2887.56 10^6bits/s over 1.001 seconds ending at 1706992898.938
Interim result: 1759.99 10^6bits/s over 1.000 seconds ending at 1706992899.938
Interim result: 1786.44 10^6bits/s over 1.000 seconds ending at 1706992900.938
Interim result: 1825.69 10^6bits/s over 1.000 seconds ending at 1706992901.938
Interim result: 1870.63 10^6bits/s over 0.601 seconds ending at 1706992902.540
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

131072 131072 131072    10.00    2106.14
```

We can see how it was many seconds before the connection was "up-to-speed" and even by ten seconds it wasn't entirely clear it was going as fast as it could. That was the congestion window growing. We can see that in another way through a packet capture, post-processed with tcptrace and viewed with the xplot.org tool. One of the types of charts produced is an outstanding data or "owin" chart which shows how much data is outstanding on the connection, waiting to be ACKed. When derived from a sending-side packet capture, this can be used as a stand-in for the congestion window computed by the sending TCP.



The vertical axis of this “owin” chart is bytes. The horizontal axis is time. The two lines of interest are the Yellow and the Red. The Yellow line is the receiver’s advertised window. The Red line is how much data was outstanding on the connection at that time. The Blue and Green lines are simply different running averages of outstanding data.

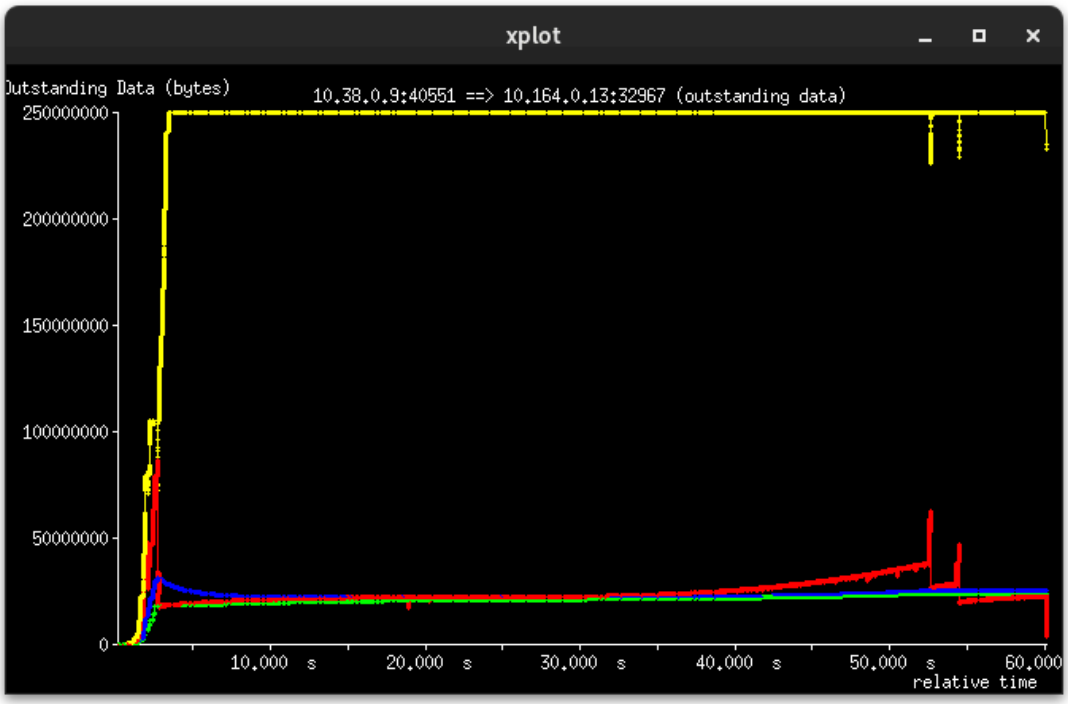
You can see that before long the Yellow line was far above the Red line. This means we were no longer being limited by the receiver’s advertised window. Instead, we were being limited by the value shown via the Red line - the proxy for Congestion Window. For the first few seconds of the test it grew exponentially as did the receive window. However, about five seconds in, something happened to bring Cubic out of the fast growth and into a slower one. The congestion window continued to grow, albeit more slowly, through the remaining five seconds of the test.

Suppose we were to increase the test time to 60 seconds?

```
receiver@amsterdam:~$ netperf -H oregon-2204 -t TCP_MAERTS -D -1.0 -l 60
MIGRATED TCP MAERTS TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.9 () port
0 AF_INET : histogram : demo
Interim result:    6.60 10^6bits/s over 1.131 seconds ending at 1706990332.216
Interim result: 1043.81 10^6bits/s over 1.000 seconds ending at 1706990333.216
Interim result: 8907.77 10^6bits/s over 1.000 seconds ending at 1706990334.216
```

```
Interim result: 8265.57 10^6bits/s over 1.159 seconds ending at 1706990335.375
Interim result: 4018.95 10^6bits/s over 1.018 seconds ending at 1706990336.393
Interim result: 2725.73 10^6bits/s over 1.000 seconds ending at 1706990337.393
Interim result: 2559.17 10^6bits/s over 1.000 seconds ending at 1706990338.393
Interim result: 2602.28 10^6bits/s over 1.001 seconds ending at 1706990339.394
Interim result: 2620.83 10^6bits/s over 1.000 seconds ending at 1706990340.395
Interim result: 2687.90 10^6bits/s over 1.000 seconds ending at 1706990341.395
Interim result: 2748.20 10^6bits/s over 1.000 seconds ending at 1706990342.395
Interim result: 2702.59 10^6bits/s over 1.017 seconds ending at 1706990343.412
Interim result: 2684.80 10^6bits/s over 1.034 seconds ending at 1706990344.446
Interim result: 2859.75 10^6bits/s over 1.000 seconds ending at 1706990345.446
Interim result: 2834.26 10^6bits/s over 1.001 seconds ending at 1706990346.448
Interim result: 2888.79 10^6bits/s over 1.000 seconds ending at 1706990347.448
...
Interim result: 1794.25 10^6bits/s over 1.055 seconds ending at 1706990365.665
Interim result: 1959.78 10^6bits/s over 1.000 seconds ending at 1706990366.666
Interim result: 1957.70 10^6bits/s over 1.000 seconds ending at 1706990367.666
Interim result: 1979.14 10^6bits/s over 1.000 seconds ending at 1706990368.666
Interim result: 1987.16 10^6bits/s over 1.000 seconds ending at 1706990369.666
Interim result: 1936.92 10^6bits/s over 1.025 seconds ending at 1706990370.691
Interim result: 1897.55 10^6bits/s over 1.057 seconds ending at 1706990371.749
...
Interim result: 2056.63 10^6bits/s over 1.000 seconds ending at 1706990386.913
Interim result: 2022.46 10^6bits/s over 1.028 seconds ending at 1706990387.940
Interim result: 1926.26 10^6bits/s over 1.064 seconds ending at 1706990389.004
Interim result: 2161.60 10^6bits/s over 1.001 seconds ending at 1706990390.005
Interim result: 2040.62 10^6bits/s over 1.000 seconds ending at 1706990391.005
Interim result: 1716.52 10^6bits/s over 0.081 seconds ending at 1706990391.086
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec
131072 131072 131072    60.00    2388.31
```

Several things to notice. First, we have a similar ramp-up, though we don't get to quite the same level before the flattening. Run to run variation. Second, even though we don't get as high early-on, as we are running for longer, the effect on overall throughput of the initial few seconds is mitigated and we have a higher overall throughput of 2388 Mbit/s rather than 2106 Mbit/s. How does the "owin" chart look?



We can see a start similar to before. There is also a spike in the Red line, which past experience has shown is a reaction to packet loss. It is as much an artifact of the way tcptrace works as anything else - tcptrace knows only that much data is unACKnowledged. It doesn't really know if it is still out on the network or not. (It isn't considering Selective ACKnowledgements) The congestion window then decreases, goes through a slow growth phase, and around 47 seconds in has another packet loss/retransmission event. That is likely what caused the drop in the interim-results towards the end.

You can also see that the Yellow line has grown to ~256 MiB. It grew to this, and no farther because the 512 MiB limit of the setting to net.ipv4.tcp_rmem was reached and Linux TCP will advertise ½ that as the receive window.

Suppose we were to use a different congestion control heuristic? We will now switch to BBR on the sender:

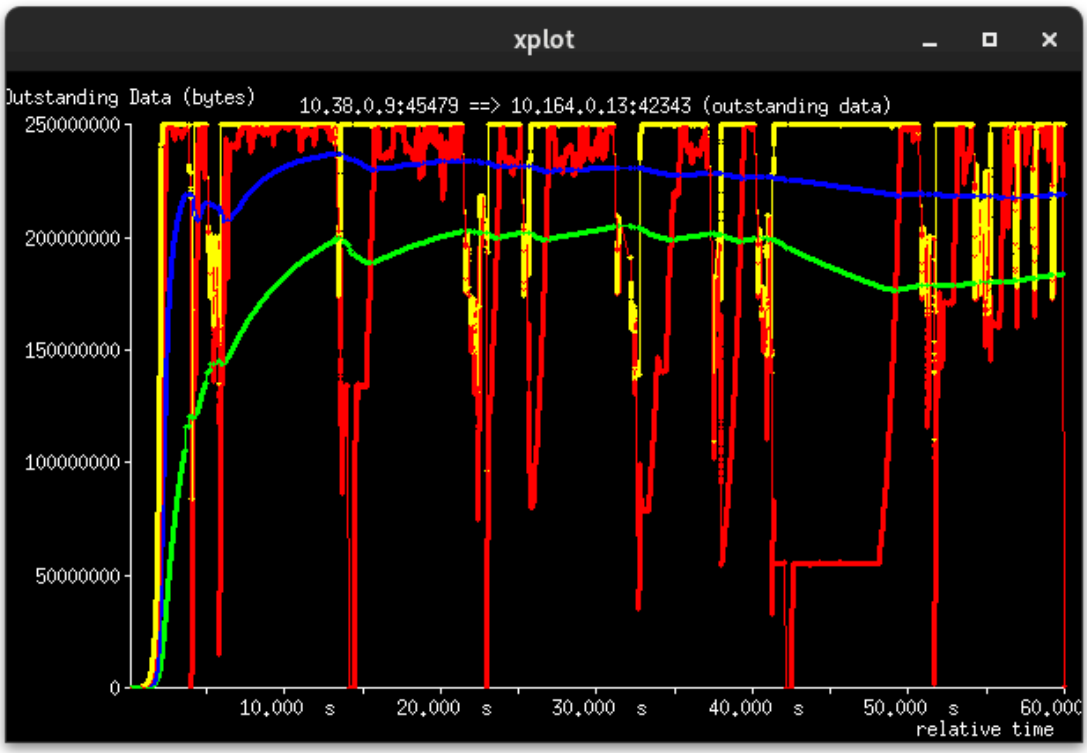
```
receiver@amsterdam:~$ netperf -H oregon-2204 -t TCP_MAERTS -D -1.0 -l 60
MIGRATED TCP MAERTS TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.9 () port
0 AF_INET : histogram : demo
Interim result:    3.80 10^6bits/s over 1.001 seconds ending at 1707438961.864
Interim result:  635.39 10^6bits/s over 1.000 seconds ending at 1707438962.864
Interim result: 5833.05 10^6bits/s over 1.241 seconds ending at 1707438964.105
Interim result: 2089.02 10^6bits/s over 1.065 seconds ending at 1707438965.170
```

```
Interim result: 9563.13 10^6bits/s over 1.000 seconds ending at 1707438966.170
Interim result: 3675.87 10^6bits/s over 1.030 seconds ending at 1707438967.200
Interim result: 7017.45 10^6bits/s over 1.000 seconds ending at 1707438968.200
Interim result: 13621.30 10^6bits/s over 1.000 seconds ending at 1707438969.200
Interim result: 10913.99 10^6bits/s over 1.016 seconds ending at 1707438970.216
Interim result: 5993.38 10^6bits/s over 1.000 seconds ending at 1707438971.216
Interim result: 8993.53 10^6bits/s over 1.000 seconds ending at 1707438972.216
Interim result: 13912.70 10^6bits/s over 1.000 seconds ending at 1707438973.216
Interim result: 8405.20 10^6bits/s over 1.024 seconds ending at 1707438974.240
Interim result: 10060.84 10^6bits/s over 1.151 seconds ending at 1707438975.391
Interim result: 13431.46 10^6bits/s over 1.000 seconds ending at 1707438976.391
Interim result: 10520.83 10^6bits/s over 1.000 seconds ending at 1707438977.391
Interim result: 4704.65 10^6bits/s over 1.012 seconds ending at 1707438978.404
Interim result: 2887.68 10^6bits/s over 1.164 seconds ending at 1707438979.568
Interim result: 9812.49 10^6bits/s over 1.000 seconds ending at 1707438980.568
Interim result: 14512.06 10^6bits/s over 1.000 seconds ending at 1707438981.568
Interim result: 8219.17 10^6bits/s over 1.000 seconds ending at 1707438982.568
Interim result: 11368.35 10^6bits/s over 1.000 seconds ending at 1707438983.568
Interim result: 13309.85 10^6bits/s over 1.000 seconds ending at 1707438984.568
Interim result: 9467.46 10^6bits/s over 1.094 seconds ending at 1707438985.662
Interim result: 7321.44 10^6bits/s over 1.004 seconds ending at 1707438986.666
Interim result: 13486.52 10^6bits/s over 1.000 seconds ending at 1707438987.666
Interim result: 11145.68 10^6bits/s over 1.024 seconds ending at 1707438988.691
Interim result: 13401.76 10^6bits/s over 1.000 seconds ending at 1707438989.691
Interim result: 5641.86 10^6bits/s over 1.020 seconds ending at 1707438990.711
Interim result: 8637.23 10^6bits/s over 1.000 seconds ending at 1707438991.711
Interim result: 11421.44 10^6bits/s over 1.000 seconds ending at 1707438992.711
Interim result: 11325.94 10^6bits/s over 1.000 seconds ending at 1707438993.711
Interim result: 7082.41 10^6bits/s over 1.167 seconds ending at 1707438994.878
Interim result: 11265.51 10^6bits/s over 1.046 seconds ending at 1707438995.923
Interim result: 11253.04 10^6bits/s over 1.014 seconds ending at 1707438996.937
Interim result: 8529.34 10^6bits/s over 1.109 seconds ending at 1707438998.046
Interim result: 11369.27 10^6bits/s over 1.049 seconds ending at 1707438999.095
Interim result: 9298.97 10^6bits/s over 1.077 seconds ending at 1707439000.172
Interim result: 7726.80 10^6bits/s over 1.032 seconds ending at 1707439001.204
Interim result: 13707.83 10^6bits/s over 1.000 seconds ending at 1707439002.204
Interim result: 13916.55 10^6bits/s over 1.000 seconds ending at 1707439003.204
Interim result: 13041.13 10^6bits/s over 1.000 seconds ending at 1707439004.204
Interim result: 10695.34 10^6bits/s over 1.000 seconds ending at 1707439005.204
Interim result: 13265.05 10^6bits/s over 1.000 seconds ending at 1707439006.204
Interim result: 13222.30 10^6bits/s over 1.000 seconds ending at 1707439007.204
Interim result: 9596.06 10^6bits/s over 1.114 seconds ending at 1707439008.318
Interim result: 8505.71 10^6bits/s over 1.093 seconds ending at 1707439009.411
Interim result: 10938.08 10^6bits/s over 1.000 seconds ending at 1707439010.411
Interim result: 13463.45 10^6bits/s over 1.000 seconds ending at 1707439011.411
Interim result: 13337.01 10^6bits/s over 1.000 seconds ending at 1707439012.411
Interim result: 9404.42 10^6bits/s over 1.000 seconds ending at 1707439013.411
```

```
Interim result: 12952.20 10^6bits/s over 1.000 seconds ending at 1707439014.411
Interim result: 11708.88 10^6bits/s over 1.164 seconds ending at 1707439015.575
Interim result: 6476.80 10^6bits/s over 1.000 seconds ending at 1707439016.575
Interim result: 13106.21 10^6bits/s over 1.000 seconds ending at 1707439017.575
Interim result: 11153.15 10^6bits/s over 1.000 seconds ending at 1707439018.575
Interim result: 11691.17 10^6bits/s over 1.106 seconds ending at 1707439019.681
Interim result: 7995.34 10^6bits/s over 1.000 seconds ending at 1707439020.681
Interim result: 12730.66 10^6bits/s over 0.182 seconds ending at 1707439020.863
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

131072 131072 131072    60.00    9718.11
```

This run is much “noisier” than the 60 second Cubic run but it does also end-up with noticeably higher throughput. How did it look in the “owin” chart?



This is a much busier chart. BBR does not react to packet losses in the same way Cubic does. It is going to look at the changes in round-trip-time and achieved throughput when deciding on a congestion window. You can see that for this run, by and large the Red line remained

above 50,000,000 bytes where Cubic never got that high. There were also times when it had indeed filled the receive window of 256 MiB.

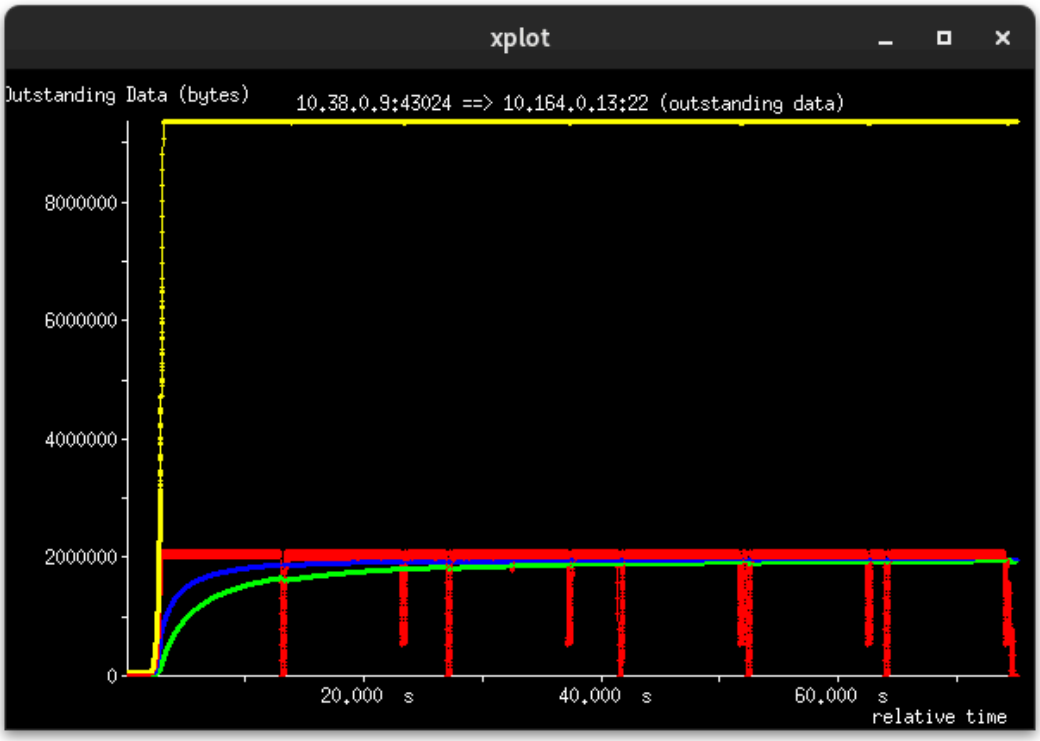
Application Window

Some applications implement their own sorts of flow control on top of TCP. A common example is ssh/scp, which has its own windowing. Unless one has the (original) High-Performance Networking SSH (HPN-SSH) patches applied at both ends, you can tune the TCP receive window to a gigabyte, have a gigabyte-worth of send socket buffer and a fully-open congestion window with no packet loss, and still have low transfer rates across a high round-trip-time path. The reason? The application-layer windowing in ssh/scp, which the (original) HPN-SSH patches address. With BBR configured on our sender, which has been shown to give us transfer rates in excess of nine gigabits per second over this path, and congestion windows greater than 50,000,000 bytes let us look at the time it takes to transfer a 1 GiB file via scp, with the (original) HPN-SSH patches. We will use a file cached in the sender's memory, and send it to /dev/null so there is no concern about storage performance:

```
amsterdam:~$ time scp 1GB.bin oregon-2204:/dev/null
1GB.bin                                       100% 1024MB  14.2MB/s   01:12

real    1m14.326s
user    0m1.681s
sys     0m1.828s
```

So, over 1 minute to transfer 1 GiB of data. Roughly speaking, 113 Mbit/s. What was the outstanding data?



Pay close attention to the scale of the y-axis as it is very different from the previous charts. You can see that the Red line showing the quantity of data outstanding was just a little bit more than 2,000,000 bytes. This is well below the congestion windows we have seen for both BBR and Cubic with "plain" TCP transfers. It is the effect of the application-layer windowing in ssh/scp. If we use the RTT of 144 milliseconds (0.14 seconds), and the apparent "effective" window of 2 MiB (16777216 bits) into:

$$Throughput \leq WindowSize \,/\, RoundTripTime$$
$$Throughput \leq 16,777,216\ bits \,/\, 0.144\ seconds$$
$$Throughput \leq 116,508,444\ bits/s$$

That is eyeball close to the actual throughput.

As for those dips in the Red line… BBR wants to know what the minimum latency of the path happens to be. That value is an important part of its calculations. As paths can change over the life of a connection, and other traffic can come and go along the path, BBR will pause for a round-trip time periodically to allow it to get an idea what the round-trip-time happens to be without its load on the path. Those weren't packet losses. While not shown here for sake of brevity, there were no retransmissions during that transfer. The slowness of the transfer

was purely the result of ssh/scp not putting enough data into the connection at one time. We've seen in the other tests that it was possible to achieve considerably higher throughput over that path.

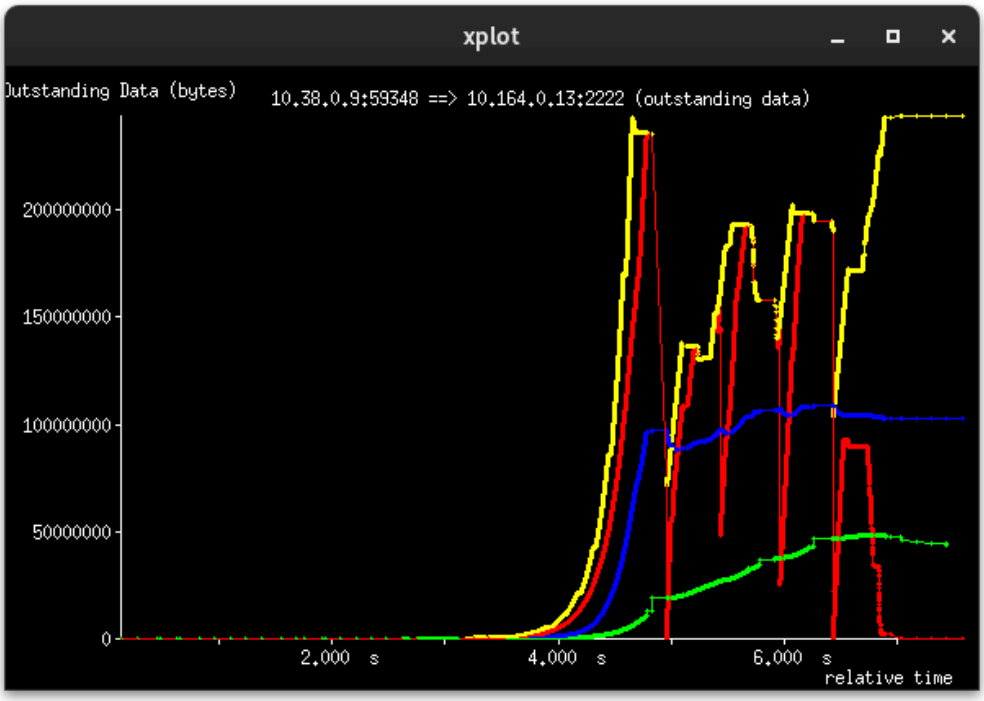
Application Window

There has been an important change in later ssh bits which have yet to be widely incorporated into Linux distros. Using them, just what sort of performance can we see?

```
oregon:~$ time hpnscp 1GB.bin 10.164.0.13:/dev/null
1GB.bin                                       100% 1024MB 231.8MB/s 459.2MB/s  00:04

real   0m7.442s
user   0m1.209s
sys    0m1.280s
```

Zounds! Now transferring that file takes less than ten seconds! How does it manage this? Let's look at the quantity of outstanding data:



Now we see the Red line going above 200 MB and getting as large as the receiver TCP's advertised window. That's 100 times as much data outstanding at one time.

You may have noticed the slightly different output from the alternate scp command. It has added an "instantaneous" rate to go with the overall. And at the end of the transfer it was reporting an overall rate of 231 MB/s with an "instantaneous" of 459 MB. Which leads us somewhat nicely to the next section... but first, you are no doubt wondering "How can I get this new scp performance?" For that, please direct your attention to https://www.psc.edu/hpn-ssh-home/ and https://sourceforge.net/projects/hpnssh/ and/or inquire of your distro provider(s) as to when they will incorporate those changes.

However, you still should not use scp as a "network" performance benchmark. It is not really suited to such things. Stick with netperf or one of the iperf variants.

Now, back to our doc...

Runtime

In situations where the round-trip-time (RTT) is large (such as between Amsterdam and Oregon) it can take a long time (relatively speaking) for a connection to get up to speed. The following is a series of runs of netperf of increasing runtime:

```
oregon:~$ HDR="-P 1"; for i in `seq 1 10`; do netperf $HDR -H amsterdam-2204 -t TCP
_MAERTS -l $i -- -O rss_size_end,lsr_size_end,send_size,elapsed_time,throughput;
HDR="-P 0"; done
MIGRATED TCP MAERTS TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.0.9 () port
0 AF_INET : histogram : demo
Remote      Local        Elapsed Throughput
Send Socket Recv Socket  Time
Size        Size         (sec)
Final       Final
2376192     2607616      1.00    3.78
358413312   304722176    2.00    333.03
500000000   433503488    3.00    2147.12
500000000   498826240    4.00    6513.04
500000000   500000000    5.00    7313.73
500000000   500000000    6.00    7234.00
500000000   500000000    7.10    6211.88
500000000   500000000    8.09    7180.11
500000000   500000000    9.17    5226.16
500000000   500000000    10.00   7524.06
```

A little noisy but as you can see, the longer the run time, the higher the overall throughput. Mostly this is how long it takes to grow the congestion window. The ending values for the SO_SNDBUF and SO_RCVBUF socket buffer sizes are included to show how they have time components to their growth as well as the congestion window.

For the sake of foolish completeness, let's re-run the runtime tests between Oregon and Iowa, where the connection has a substantially lower average RTT (38 ms vs 140). We leave all other settings the same.

```
oregon:~$ HDR="-P 1"; for i in `seq 1 10`; do netperf $HDR -H iowa-2204 -t
TCP_MAERTS -l $i -- -O
rss_size_end,lsr_size_end,send_size,elapsed_time,throughput;HDR="-P 0"; done
MIGRATED TCP MAERTS TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.128.0.2 () port
0 AF_INET : histogram : demo
Remote      Local        Elapsed Throughput
Send Socket Recv Socket  Time
Size        Size         (sec)
Final       Final
500000000   440402688    1.00    5648.80
500000000   500000000    2.07    7970.94
500000000   500000000    3.00    8931.29
500000000   500000000    4.00    7451.06
500000000   500000000    5.02    11530.87
500000000   500000000    6.00    9534.63
500000000   500000000    7.01    7855.20
500000000   500000000    8.00    10544.44
500000000   500000000    9.00    10951.82
500000000   500000000    10.00   12096.49
```

You can see not only do we achieve higher throughput thanks to the reduced Round-Trip-Time, we also get to the higher throughput sooner. To further underscore that point, and augment the runtime discussion, let's run a slightly different netperf test. One where it emits interim results every second over the life of a 20-second test:

```
oregon:~$ netperf -H amsterdam -t TCP_MAERTS -l 20 -D -1 -- -O
rss_size_end,lsr_size_end,send_size,elapsed_time,throughput
MIGRATED TCP MAERTS TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.164.0.78 ()
port 0 AF_INET : histogram : demo
Interim result:    3.03 10^6bits/s over 1.000 seconds ending at 1718235875.189
Interim result:  408.25 10^6bits/s over 1.000 seconds ending at 1718235876.189
Interim result: 10128.56 10^6bits/s over 1.000 seconds ending at 1718235877.189
Interim result: 12269.18 10^6bits/s over 1.000 seconds ending at 1718235878.189
Interim result: 12778.22 10^6bits/s over 1.000 seconds ending at 1718235879.189
Interim result: 13502.52 10^6bits/s over 1.000 seconds ending at 1718235880.189
Interim result: 13242.04 10^6bits/s over 1.000 seconds ending at 1718235881.189
Interim result: 13304.82 10^6bits/s over 1.000 seconds ending at 1718235882.189
Interim result: 8024.42 10^6bits/s over 1.153 seconds ending at 1718235883.341
Interim result: 8350.97 10^6bits/s over 1.000 seconds ending at 1718235884.341
Interim result: 13284.74 10^6bits/s over 1.000 seconds ending at 1718235885.341
Interim result: 13101.24 10^6bits/s over 1.000 seconds ending at 1718235886.341
```

```
Interim result: 8019.36 10^6bits/s over 1.077 seconds ending at 1718235887.419
Interim result: 11916.23 10^6bits/s over 1.000 seconds ending at 1718235888.419
Interim result: 13102.68 10^6bits/s over 1.000 seconds ending at 1718235889.419
Interim result: 13581.05 10^6bits/s over 1.000 seconds ending at 1718235890.419
Interim result: 13465.39 10^6bits/s over 1.000 seconds ending at 1718235891.419
Interim result: 13468.71 10^6bits/s over 1.000 seconds ending at 1718235892.419
Interim result: 13087.14 10^6bits/s over 1.000 seconds ending at 1718235893.419
Interim result: 13428.06 10^6bits/s over 0.769 seconds ending at 1718235894.188
Remote      Local        Elapsed Throughput
Send Socket Recv Socket  Time
Size        Size         (sec)
Final       Final
500000000   500000000    20.00   10860.93
```

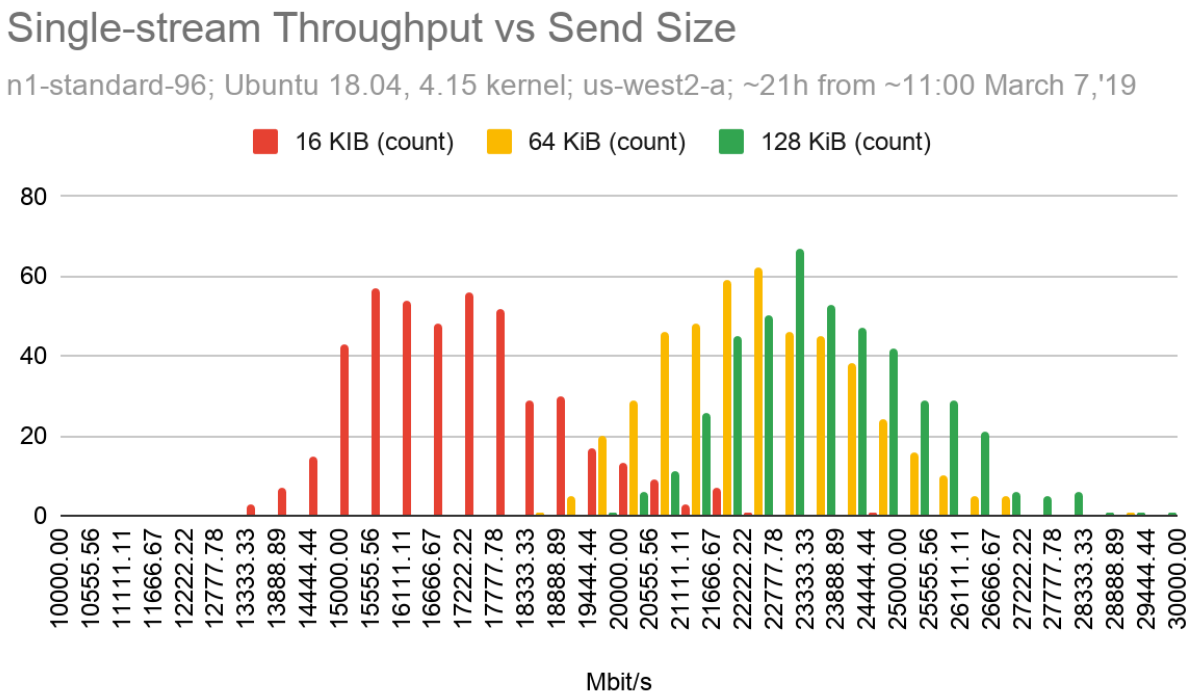
We can see, as with the enhanced scp output, how at the end we were going much faster than our overall. How long it would take for the overall to asymptotically converge to the instantaneous is left as an exercise to the reader :)

Send Size

A netperf TCP_STREAM/TCP_MAERTS test picks a default send size based on the size of the socket send buffer (SO_SNDBUF) at the time the data socket is created. Under Linux this is usually 16 KiB, though that can change with the middle value for the net.ipv4.tcp_wmem sysctl. This is generally fine in many situations, but once the desired throughput gets much above 16-20 Gbit/s this small a send size can lead to saturating a CPU on the sending side. To choose a larger send size, the test-specific -m option should be used:

```
netperf -H target -t TCP_STREAM -- -m 128K
netperf -H target -t TCP_MAERTS -- -m ,128K
```

That will have netperf use a send size of 128 KiB (131072 Bytes). Pay note to the comma in the command line for the TCP_MAERTS test. The effect can be non-trivial:



You can see how a single-stream with 16KiB send calls could often achieve the old maximum egress throttle of 16 Gbit/s, but would not hit the current maximum egress throttle of 32 Gbit/s. Using 64KiB or 128 KiB gets single-stream performance much closer to that. Add in a few more streams to hit the throttle.

MTU

The final variable that we'll discuss here is the maximum transmission unit (MTU). This is the size of the largest packet that a network is capable of carrying (data bytes + packet headers). By increasing the size of the MTU, we increase the ratio of data bytes to header bytes in a packet, giving us substantially lower overhead. In Google Cloud, we can increase the MTU up to a maximum of 8896 bytes. To use a larger MTU, we must configure it at the time of VPC creation and then also make sure all VMs on the network are configured to use the same MTU. If you have multiple connected VPCs, it is best to set them all to use the same MTU to avoid issues.

Because much of the Internet uses an MTU of 1500 bytes, if we want to send traffic externally, it may be best to keep the packet size at 1500 bytes to prevent packet loss or fragmentation. More information about MTU settings can be found in our documentation.

Now with the caveats out of the way, let's see how our Amsterdam to Oregon example performs with an MTU of 8896 and using BBR for congestion control.

```
phanekham@amsterdam:~$ netperf -H oregon-2204 -t TCP_MAERTS -D -1.0 -l 60
MIGRATED TCP MAERTS TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to 10.138.15.210 ()
port 0 AF_INET : histogram : demo
Interim result:   10.23 10^6bits/s over 1.003 seconds ending at 1707457656.857
Interim result: 1634.57 10^6bits/s over 1.000 seconds ending at 1707457657.857
Interim result: 13261.64 10^6bits/s over 1.000 seconds ending at 1707457658.857
Interim result: 12657.40 10^6bits/s over 1.000 seconds ending at 1707457659.857
Interim result: 12620.67 10^6bits/s over 1.000 seconds ending at 1707457660.857
Interim result: 12655.79 10^6bits/s over 1.000 seconds ending at 1707457661.858
Interim result: 12672.82 10^6bits/s over 1.000 seconds ending at 1707457662.858
Interim result: 12745.26 10^6bits/s over 1.000 seconds ending at 1707457663.858
Interim result: 12975.00 10^6bits/s over 1.000 seconds ending at 1707457664.858
Interim result: 12862.55 10^6bits/s over 1.000 seconds ending at 1707457665.858
...
Interim result: 12569.84 10^6bits/s over 1.000 seconds ending at 1707457677.858
Interim result: 12552.13 10^6bits/s over 1.000 seconds ending at 1707457678.858
Interim result: 12594.72 10^6bits/s over 1.000 seconds ending at 1707457679.858
Interim result: 12579.75 10^6bits/s over 1.000 seconds ending at 1707457680.858
Interim result: 13000.21 10^6bits/s over 1.000 seconds ending at 1707457681.858
Interim result: 13364.22 10^6bits/s over 1.000 seconds ending at 1707457682.858
Interim result: 13451.18 10^6bits/s over 1.000 seconds ending at 1707457683.858
...
Interim result: 13441.68 10^6bits/s over 1.000 seconds ending at 1707457709.859
Interim result: 13429.56 10^6bits/s over 1.000 seconds ending at 1707457710.859
Interim result: 13438.45 10^6bits/s over 1.000 seconds ending at 1707457711.859
Interim result: 8545.42 10^6bits/s over 1.093 seconds ending at 1707457712.952
Interim result: 13544.99 10^6bits/s over 1.000 seconds ending at 1707457713.952
Interim result: 13484.06 10^6bits/s over 1.000 seconds ending at 1707457714.952
Interim result: 13458.37 10^6bits/s over 0.903 seconds ending at 1707457715.855
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

131072 131072 131072    60.00    12508.57
```

We can see that we have achieved the highest throughput yet of 12.5 Gbits/sec. We can also see that there is generally lower variability than with our BBR test at an MTU of 1460. So while not always practical to use a higher MTU, it can provide substantial benefit to throughput.

Acknowledgements

We would like to thank our colleague Neal Cardwell for his feedback.