Google Cloud

A Brief Look at Round Trip Time

There Is More To It Than The Network

Rick Jones

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

Introduction

Round Trip Time, or **RTT** as it is often called, is a commonly-tracked metric when networks are involved. TCP stacks will measure it as part of deciding when they must presume segments lost and retransmit them. Networked applications will measure the time between sending a request and receiving a response, and report that as their RTT. Together with information about packet loss rates it is the primary output of the various forms of the venerable "`ping`" command. Different "network benchmarks" will measure RTT either by reporting TCP's measure or by timing their own request/response pairs.

Google Cloud will also report RTTs based on what it sees in VM traffic it samples.

And of course, people will "blame the network" when they see a high RTT value :)

This writeup will attempt to describe how all those RTT measurements behave so those of you looking at those reports can interpret them correctly.

Summary

The way TCP measures RTT can be influenced by receive-side application behavior and so is not "really" a network-level RTT[1]. Since it does not include time spent retransmitting segments it will not directly reflect what applications may see in the presence of packet loss. Google Cloud's metrics for RTT, which are based on sampled TCP traffic and measure RTT in the same way, are similar in that regard.

Request/Response timing measurements by the likes of a netperf TCP_RR test, being fully "application-to-application" (i.e. "above" TCP) will be influenced by virtually all the same things in a TCP-level RTT measurement, with the added influence of packet loss. "Ping" (ICMP Echo Request/Response) will not have a receive-side application component, and will not "retransmit" on lost traffic as such, and so will be closest to a "network level" round-trip-time measurement. However, all of these are influenced by the intervals between probes.

[1] You might think of it as more "Time to Get an ACK"

How RTTs Are Measured

TCP

While there can be more nuance than we want to get into here, RTT measurement by TCP is conceptually pretty simple. TCP measures the time between when it sends a given sequence number and when it receives an ACKnowledgement for that sequence number. The key of course being "when it receives an ACKnowledgement for that sequence number." And this is where receiving application behavior and performance comes into play.

TCP ACKnowledgements are necessary overhead. Being overhead means TCP wants to minimize their number. Being necessary means TCP has to send at least a few. So, broadly speaking, TCP will employ the following heuristics when deciding when to send an ACK:

1. Is there data to be sent back in the other direction? If yes, piggyback the ACK on that data segment.
2. Has the receiving application read enough data out of the socket to warrant sending a Window Update back in the other direction? If yes, piggyback the ACK on that Window Update.
3. Wait until either 1, or 2 are true, or the Standalone[2] ACK Timer expires. Send the ACK then.

Under Linux, the Standalone ACK timer is 40 milliseconds. Under Windows it appears to be 200 milliseconds in older versions, and... complicated in later versions. Either 10, 40 or 50 ms depending on Client vs Server and Internet versus DataCenter profiles. There are additional heuristics within the Linux TCP stack where it will decide to ACK immediately, but we won't get into them here except perhaps in passing.

What this means is that it can be as long as the actual network-level round-trip-time ***plus the minimum of the receiving application's service time and the standalone ACK timer*** before a sending TCP receives an ACKnowledgement and takes that and sticks it into its RTT calculations.

A further "wrinkle" in this mix is TCP Tail Loss Probing. When a TCP sender sends a train of segments it can reasonably assume will have generated timely ACKnowledgements, the sending TCP may retransmit the "tail" (last segment) of that train sooner than its minimum retransmission timeout (TCP_RTO). The idea being to ascertain if perhaps the tail of the packet train was lost. If the tail loss probe arrives at the receiver, it will likely trigger an

[2] Some stacks will refer to it as the Delayed ACK Timer.

immediate ACK+SACK[3] by virtue of either being duplicate, or data received out-of-order (defining a "gap" in the received sequence space). The timer for this tail loss probe will be shorter than either the sender's TCP_RTO or the receiver's standalone ACK timer. Experience has shown it to be often on the order of 4 ms with a Linux stack[4], though that isn't a guarantee.

What also is not a guarantee is that what the sending TCP has sent can be reasonably assumed to have generated a timely ACKnowledgement. Not all traffic patterns will have tail-loss probing.

Further, TCP will not use ACKs for retransmitted data segments in its RTT measurements. It will use only "clean" ACKs for un-retransmitted segments.

What all this means is that what TCP will report as an RTT will be influenced by heuristics and application behavior - sometimes as much if not more so than by the actual "network-level" Round-Trip-Time. And will not reflect the time needed to retransmit data.

Google Metrics

The Google Metrics for RTT work in a manner very similar to TCP, only with samples of the traffic rather than all of it. When the sampler sees a TCP data segment leaving the VM, the sampler will note a TCP sequence number and will generate an RTT sample if it also samples an ACKnowledgement for that sequence when the ACK arrives for that VM. Apart from not including time spent in the VM for which the traffic is sampled, it will measure the same things TCP running in that VM will. One exception however is that the Google Metrics do not have a good idea whether an ACK is for a retransmitted data segment or not and so _may_ include that in the RTT measurement.

Netperf TCP_RR

What a netperf TCP_RR test measures for an "RTT" is different from what TCP measures. It measures the time just before calling "send()" for a request and just after returning from "recv()" for the last byte of the response. Since TCP operating under the proverbial covers might have to recover from packet losses in order to deliver the bytes of either the request or response, the "round-trip time" measured by netperf will include the time it takes to do that. It also includes all the time to get to/from/through TCP itself in addition to everything else. For that reason, even when a request and a response are each a single TCP segment, it can

---

[3] Selective ACKnowledgement - a way for TCP to acknowledge data segments which are received out-of-order rather than having to wait until the "holes" are filled for a cumulative ACK.
[4]Under Linux the TLP timer should be roughly 2*srtt + 2 jiffies. (Those of a programming bent can see tcp_schedule_loss_probe() for details.)

report an “RTT” longer than what TCP calculates, or what might be seen at the Google Metrics sampler(s).

IPerf

IPerf/iPerf3 simply reports what TCP computes for RTT. It does not have a netperf TCP_RR-like test.

Ping

The ICMP Echo Requests generated by a “ping” command are usually sent from user space, and in that sense then are “similar” to say netperf sending a request. However, as there is (usually) no receiving application, there is no receiving application influence. And no equivalent to the standalone ACK or tail-loss probe timers. The ICMP Echo Request is seen by “IP” at the destination and an ICMP Echo Reply is generated immediately[5].

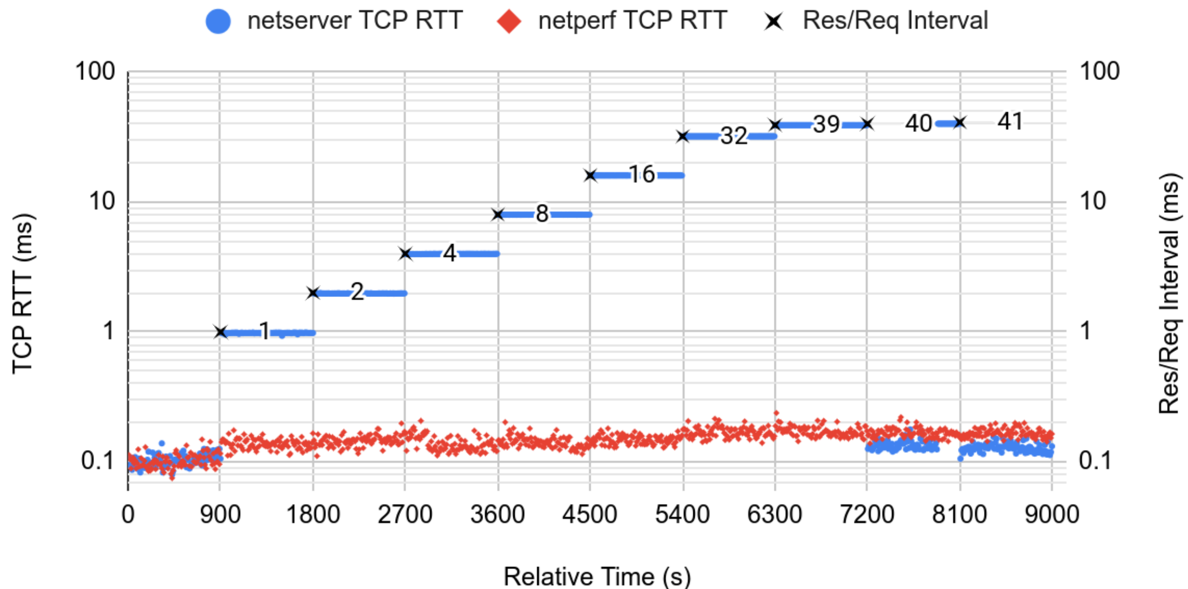
Results

Here we take a look at various reported RTTs as computed/measured by TCP, Google Metrics, and netperf.

What TCP Saw

What about this receive side application influence? Well, let’s take a look at a chart showing the RTT computed by TCP on each side of a netperf TCP_RR test, where we alter the time between receipt of a response by netperf and when it sends the next request:

[5] Well, within the time it takes to get to IP at the destination. For example, when the destination is a VM, while the IP code running in the guest VM will respond immediately, the VM itself can be thought of as being like an application, and it may take some time to get running if it was sleeping or if other VMs were consuming CPU. And of course, there can be ICMP rate limiting at the destination - but that will manifest as loss rather than added round-trip time, and that is beyond the scope of this writeup :)

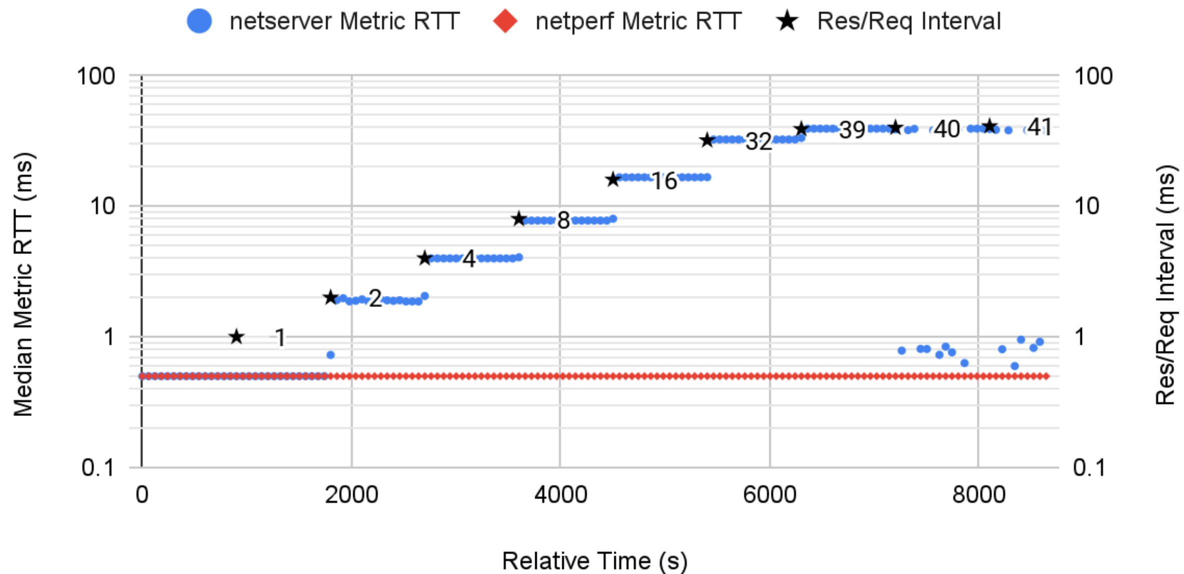


What we see here are the TCP-measured RTTs for both sides of the connection(s) - netperf and netserver. When the netserver side application receives a complete request it will immediately send a response, so the ACK of the request will be piggybacked with the response without any additional delay. This then is why the TCP RTT on the netperf side is as low as it is.

As the Response/Request (Res/Req) Interval is increased from 0 to 41 milliseconds it increases the time between when netperf receives a response and when it sends the next request. So, it is longer and longer before there is data to go back the other way (ie the next request), and we can see how this results in TCP at the netserver side reporting a longer and longer "RTT." This continues until that interval reaches and exceeds the Linux TCP's Standalone ACK timer. It is at that point when some of these additional Linux heuristics kick-in and it starts to generate immediate ACKnowledgements. The timers and such running in netperf and in TCP are independent of one another, so there can be some differences in which "wins" a given race. That may be why we see both immediate and after-timer ACKs at the 40ms Res/Req Interval level.

What Google Metrics Saw

Here next we have a chart of the Google sampled RTT metrics from the same netperf runs:



The sharp-eyed will notice how the netperf Metric RTT is reported as 0.5 milliseconds whereas the TCP computed RTT for the netperf VM was between 0.1 and 0.2 milliseconds. This is an artifact of how the Google Metrics “bucket” the samples for networking.googleapis.com/vm_flow/rtt with the smallest bucket being for 0-1 millisecond. Virtually all the samples from the “netperf” VM became counts in that bucket. Thus the computed median (aka 50th percentile) of 0.5 milliseconds. This brings up another consideration when looking at RTT metrics - how they are aggregated will affect the values being reported.

What Netperf Saw

Netperf too keeps a histogram of the response times it sees. It is a “log linear” histogram which starts at microseconds and has ten buckets at each order of magnitude. So, ten buckets for microseconds, ten buckets for tens of microseconds, ten buckets for 100s of microseconds, and so on and so forth[6]. Via the “output selectors” it can be asked to report various percentiles computed from that histogram, which we present here in table form:

Milliseconds	Microseconds				
Res/Req Interval	MIN_LATENCY	MEAN_LATENCY	P50_LATENCY	P90_LATENCY	P99_LATENCY
0	60	101.42	95	133	197
1	80	149.93	141	197	277
2	85	161.53	153	210	293
4	85	155.68	146	206	299
8	93	158.57	149	209	300
16	101	171.23	162	226	324
32	115	193.77	186	249	342
39	114	198.06	190	253	352
40	109	197.61	189	259	353
41	109	192.88	186	246	334

We can see response times increasing somewhat as the interval increases, but they do not increase at all like the TCP and Google Metrics do for the netserver side. The increases we see here come from the underlying effect of the interval between probes on the measured response time.

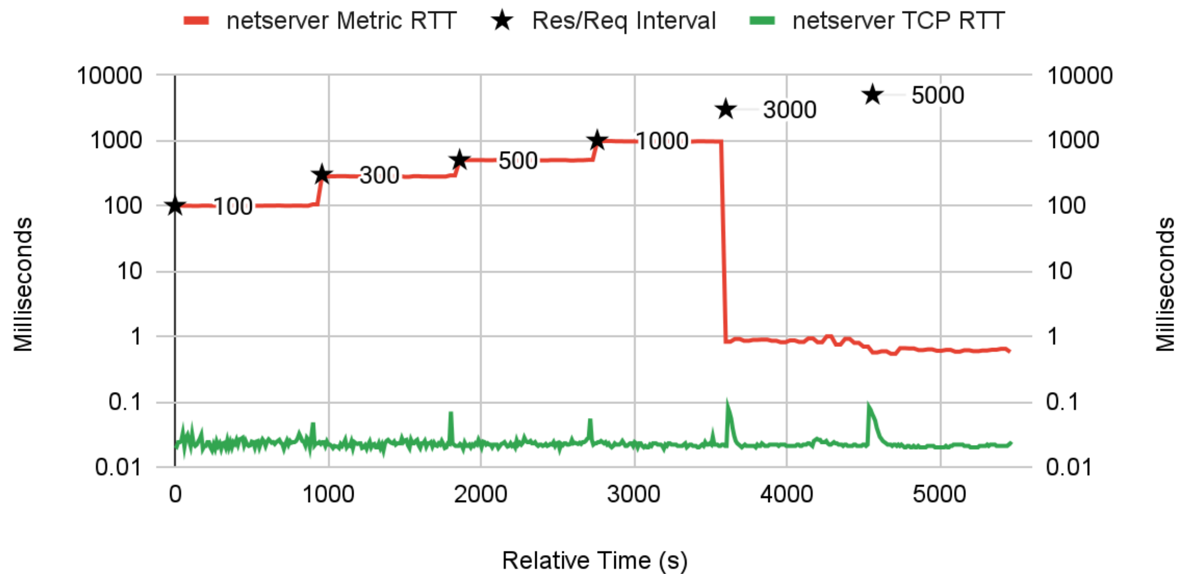
Beyond The Standalone ACK Timer

To this point, we’ve asserted a (rough) equivalence between what a TCP endpoint might compute for Round-Trip Time, and what Google Metrics might report. And the results we’ve seen thus far support that. However… things can begin to diverge after the responding application’s response time goes beyond the responding application’s TCP’s standalone ACK timer.

Recall that the Google Metrics for RTT are based on sampled traffic. Unlike TCP, it will not look at all the packets to/from a VM, only a (possibly very) small subset of them. When the

[6] Actually, it uses still-finer granularity internally, but the histogram it can be asked to report is based on that “log linear” progression.

responding application's response time goes beyond the standalone ACK timer, the packet(s) which the sampler sees to compute an RTT may be the data segments of the response rather than the standalone (or immediate) ACKnowledgement from TCP. So, Google Metrics can report RTTs well beyond the length of the TCP Standalone ACK timer. And indeed, that is what we can see in the following chart:



We take the interval between requests through 100, 300, 500, 1000, 3000, and 5000 milliseconds. You can see the TCP-measured RTT is largely unchanged across the entire range. TCP is seeing the immediate ACKnowledgements being sent and computing its RTT value accordingly. The packet sampler however is not seeing the immediate ACKnowledgements, or at least not often enough for those to dominate the calculation. At least not up through 1000 milliseconds between requests. Rather it is seeing the data segments of the subsequent requests. That then is what goes into the Metric's RTT calculation. In effect it is measuring application-level response times rather than network round-trip times.

Somewhere at/around 3000 milliseconds between requests this behavior changes for the sampler[7]. But we still have an "RTT" measurably higher than the actual, network-level RTT. The whys and wherefores of that beyond bucketing are left for another day. 🙂

Setup

Initial

The results presented in this writeup were from a pair of e2-standard-32 VMs running Ubuntu 22.04 in Google's southamerica-west1-a Availability Zone. They employed the virtio_net vNIC and ran the 5.15.0-1025-gcp Linux kernel. No placement group was employed, and the sysctl settings were at their defaults for the 5.15.0-1025-gcp Linux kernel.
Traffic was VM to VM same-zone using Internal IPs.

TCP computed RTTs are as reported via the "`ss -i`" command, which was sampled every ten seconds on both sides as the netperf commands were run. The requests being sent by netperf were a single byte and so a single TCP segment. The responses being returned by netserver were two bytes and so also a single TCP segment. The same TCP four-tuple was used for each measurement. This greatly simplified gathering the TCP computed RTTs via the "`ss -i`" command.
The networking.googleapis.com/vm_flow/rtt median values were obtained via the metrics API explorer with a curl equivalent stated to be:

```
curl \
'https://monitoring.googleapis.com/v3/projects/[YOUR_PROJECT]/timeSeries?aggregation.alignmentPeriod=60s&aggregation.crossSeriesReducer=REDUCE_MEAN&aggregation.perSeriesAligner=ALIGN_PERCENTILE_50&filter=metric.type%3D%22networking.googleapis.com%2Fvm_flow%2Frtt%22%20resource.type%3D%22gce_instance%22%20metadata.system_labels.%22name%22%3Dmonitoring.regex.full_match(%22[YOUR_VM_NAME]%22)%20metric.label.%22remote_region%22%3D%22[YOUR_REGION]%22%20metric.label.%22remote_location_type%22%3D%22CLOUD%22&interval.endTime=2022-12-05T17%3A04%3A00-08%3A00&interval.startTime=2022-12-05T14%3A34%3A00-08%3A00&fields=timeSeries.metric%2CtimeSeries.points&key=[YOUR_API_KEY]' \
  --header 'Authorization: Bearer [YOUR_ACCESS_TOKEN]' \
  --header 'Accept: application/json' --compressed
```

[7] Based on some quick, ad-hoc testing, one of the variables is the quantity of data being sent - one quick test where that was doubled from here showed a Metric Median RTT of slightly more than 10 milliseconds. Increasing the quantity of data further shifts the ratio of bare ACKs to data segments in favor of data segments.

The minimum granularity available for those values is 60 seconds. And of course, you would alter your start and end times accordingly.

The netperf tests were launched via:

```
for i in -1 1 2 4 8 16 32 39 40 41;
do
  netperf -H <netserver_vm_ip> $HDR -t TCP_RR -B "`date +%s`,${i}" -l 900 -w ${i}m -b 1 -- -r 1,2 -o result_brand,MIN_LATENCY,MEAN_LATENCY,P50_LATENCY,P90_LATENCY,P99_LATENCY -P 65432;
  HDR="-P 0";
done
```

Where “-1” passed to the “-w” option means no waiting between receipt of response and sending of next request.

Beyond The Standalone ACK Timer Configuration

For this configuration, we still use a pair of VMs, e2-standard-32s with virtio_net and Ubuntu 22.04. The kernels are a bit newer.

This time, rather than single-segment requests and responses, we have the netperf side send a two-segment request. This increases the odds the sampler will pick one or the other of the two data segments rather than the earlier bare/standalone ACK. Also, since the inter-request intervals were much larger than before, to keep the packet rates roughly the same as the intervals are increased, the number of parallel streams is increased with each increase in the inter-request interval.

```
for j in 1 3 5 10 30 50;
do
  END=`expr ${j} \* 15`
  for i in `seq 0 ${END}`;
  do
    netperf -P 0 -t TCP_RR -H boulder -l 900 -b 1 -w `expr ${j} \* 100`m -- -r 1409,2 &
    sleep 0.05;
  done;
  wait;
done
```

This will then run with inter-request intervals of 100, 300, 500, 1000, 3000, and 5000 milliseconds, and increase the number of parallel netperf sessions running as it goes.

Acknowledgements

The author would like to thank Neal Cardwell for his review and mention of the computation of the Tail-Loss Probe (TLP) timer under Linux. He would like to thank Kevin Hogan for the suggestion to increase the netperf request size to more reliably reproduce seeing the ACK-carrying next request rather than the immediate ACK when going beyond the standalone ACK timer. He would also like to thank Derek Phanekham for editing and updating this document for public release.