Google Cloud

Measuring Aggregate Packets Per Second with netperf

Sometimes There Is More To Life Than Mbits/s But This Method Will Work With a Bulk Transfer

Rick Jones

Introduction

One of the most commonly used measures of network performance for a system is unidirectional, bulk transfer - iperf3, netperf TCP_STREAM, etc. It is usually straightforward and easy to do, and easily compared with things like NIC speed. However, not all systems service bulk-transfer workloads. Sometimes it isn't how many megabits per second can be sent but how many transactions per second which matters.

This write-up will describe one way one can use netperf to measure aggregate transactions per-second and so at least one variation on aggregate packets per second for a system.

Summary

1. Install netperf on each system:
    a. Ensure gcc, make, automake, texinfo and python-rrdtool are installed. That can be python3-rrdtool with an updated version of post_proc.py
    b. Bring a netperf source tree to the systems
    c. cd to the top of the netperf source tree
    d. run ./autogen.sh
    e. ./configure --enable-burst --enable-demo --enable-histogram
    f. make # sudo make install if you prefer
    g. src/netserver # start the netserver on the load generator systems
2. On the system to be the one under test, cd to doc/examples/ in the netperf source tree
3. Ensure that runemomniaggdemo.sh and find_max_burst.sh have the execute bit set
    a. chmod +x runemomniaggdemo.sh find_max_burst.sh
4. Edit runemomniaggdemo.sh to:
    a. Enable aggregate Transactions Per Second (tps) tests - DO_RRAGG set to '1'
    b. Disable the other tests - DO_mumble set to '0'
    c. Set DURATION to your desired time length for each data point - eg 60 or 120 (seconds)
5. Add the location of the find_max_burst.sh script to your $PATH variable
    a. export PATH=$PATH:.
6. Edit/create a "remote_hosts" file like the following, with the names or IP addresses of your load generators:
```
REMOTE_HOSTS[0]=lg1
REMOTE_HOSTS[1]=lg2
REMOTE_HOSTS[2]=lg3
REMOTE_HOSTS[3]=lg4
NUM_REMOTE_HOSTS=4
```
7. ./runemomniaggdemo.sh # execute the script
8. ./post_proc.py --intervals netperf_tps.log # script relies on python_rrdtool

9. Enjoy the results. There will also be a chart in “netperf_tps_overall.svg” for those who prefer pictures over text

If you are compiling elsewhere and just bringing binaries rather than putting a netperf source tree on every system, you will need to bring the netperf binary to the system under test, along with the runemomniaggdemo.sh and find_max_burst.sh scripts and a remote_hosts file. If you wish to post-process results there, bring post_proc.py as well. The systems acting as load generators need only the netserver binary and it started on them.

Theory of Operation

There are many different ways one can go about running aggregate tests with netperf. Each has advantages and drawbacks. One way is to simply run N copies of netperf in a shell loop and summarize the results. For example we could run four (4) netperf TCP_RR tests between our system under test and a load generator:

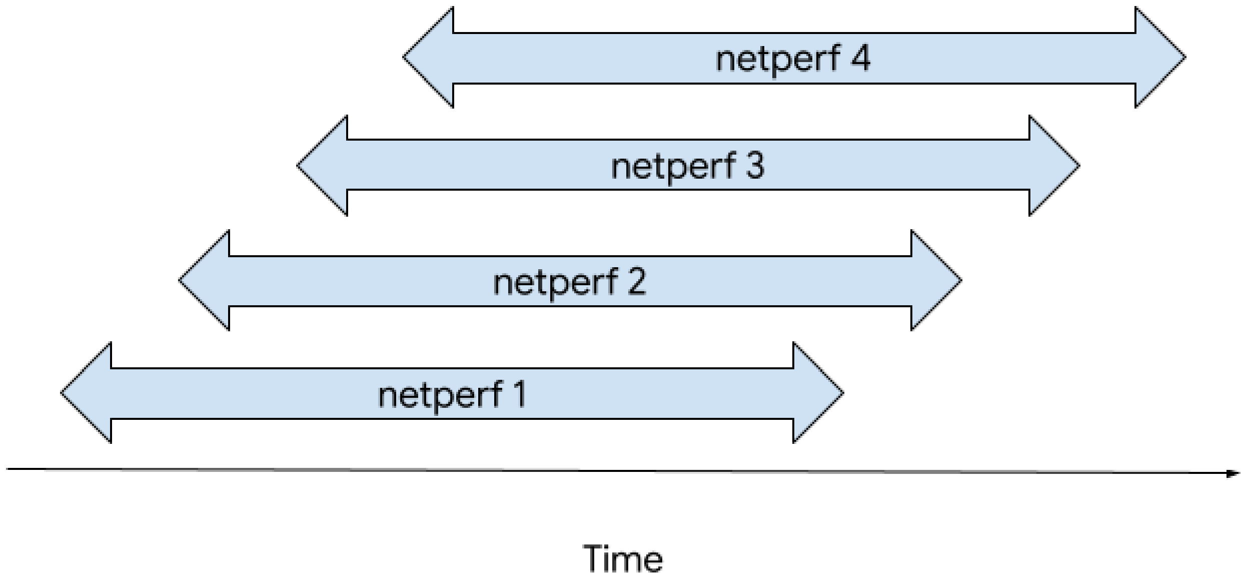
```
jonesrick@sut:~/netperf-2.7.1$ for i in `seq 1 4`
do
  netperf -t TCP_RR -H lg1 -P 0 &
done
...
jonesrick@sut:~/netperf-2.7.1$ 16384  87380  1        1       10.00    18929.37
65536  87380
16384  87380  1        1       10.00    18229.11
65536  87380
16384  87380  1        1       10.00    19218.99
65536  87380
16384  87380  1        1       10.00    18666.15
65536  87380


# And then sum-up the transactions/s to arrive at a result.  Say with a simple one
liner at the shell:

jonesrick@sut:~/netperf-2.7.1$ awk '{sum += $6}END{print "sum",sum}'
16384  87380  1        1       10.00    18929.37
65536  87380
16384  87380  1        1       10.00    18229.11
65536  87380
16384  87380  1        1       10.00    19218.99
65536  87380
```

```
16384  87380  1        1       10.00    18666.15
65536  87380
sum 75043.6
```

(Using $6 is a cheap way to "ignore" the second line of each netperf's results and so avoid having to be selective in cut-and-paste)

Ninety-nine times out of ten (sic[1]) that will be fine for small-scale tests - not too many parallel streams. But it suffers from the threat of skew error. We are assuming all four netperfs started at virtually the same time, and they stopped at virtually the same time, which means they all were running at the same time.

However, it is possible that they didn't start and/or finish at the same time - those times could be skewed. That would mean that not all the netperfs were always running at the same time and one cannot simply sum their results.



One way to address skew is to simply make the run times much longer so start/stop skew is essentially epsilon. That can mean rather long run times.

[1] Never learn expressions from high-school athletics coaches. And be wary of learning humor from high-school Physics teachers. The half-octopus rule is now in effect.   Forewarned is forearmed... :)

Another way to address skew would be for each of the streams of tests to actually be three sequential runs of netperf, with the length of the first and last of each long enough to be longer than any skew, and their results ignored. Then one could sum the results of the middle netperf in each stream:

```
jonesrick@sut:~/netperf-2.7.1$ for i in `seq 1 4`
> do
> (netperf -t TCP_RR -H lg1 > /dev/null; netperf -t TCP_RR -H lg1 -P 0; netperf -t
TCP_RR -H lg1 > /dev/null) &
> done
```

The downside to this is the added time to completion of the test. In the case above, instead of taking 10 seconds for our “datapoint” it took 30 seconds. More generally, if you know the skew in advance, it means running for runtime + 2*skew.

When launching netperfs on the same system, there is usually not too much skew - at least not until one has many of them or one runs something like UDP_STREAM and really hammers the system. But suppose we want to start netperfs on several systems. There we will have the added effects of initiating netperf on other systems, and the possibility of skew error is more severe.

Ideally, we’d like to be able to tell netperf to start generating load, then be able to tell it when to start measuring results, when to stop measuring, when to stop generating load and then report only the measurement interval. But netperf is a simple benchmark. It runs. It stops. That’s it. It’s bliss.

So, what can we do?

One other option would be to just start the netperfs, and then look at other sources for how fast things were going. Take snapshots of ifconfig or netstat statistics at a given cadence and use those for the results. That will work, but requires more coordination, and we can get something very similar from netperf.

When building netperf from source, one can enable a mode called “demo mode” whereby netperf can be asked to emit interim results at a desired rate.

```
./configure --enable-demo
make netperf
```

With that in place, and a global "-D" option ("Demo output") added to the command-line, netperf output becomes:

```
jonesrick@sut:~/netperf-2.7.1$ netperf -t TCP_RR -H lg1 -D -1.0
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
lg1.c.netperf-163
618.internal () port 0 AF_INET : histogram : spin interval : demo : first burst 0
Interim result: 18076.95 Trans/s over 1.000 seconds ending at 1542304520.045
Interim result: 17869.57 Trans/s over 1.000 seconds ending at 1542304521.045
Interim result: 17842.14 Trans/s over 1.000 seconds ending at 1542304522.045
Interim result: 18115.64 Trans/s over 1.000 seconds ending at 1542304523.045
Interim result: 18570.33 Trans/s over 1.000 seconds ending at 1542304524.045
Interim result: 18254.73 Trans/s over 1.000 seconds ending at 1542304525.045
Interim result: 18180.56 Trans/s over 1.000 seconds ending at 1542304526.045
Interim result: 17871.89 Trans/s over 1.000 seconds ending at 1542304527.045
Interim result: 18457.91 Trans/s over 1.000 seconds ending at 1542304528.045
Interim result: 18473.11 Trans/s over 1.000 seconds ending at 1542304529.045
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec
16384  87380  1        1       10.00    18170.89
65536  87380
```

You can use different lengths of time with the -D option. The negative for the value tells netperf to be more aggressive about actually hitting the desired interval by making many more checks for the time[2].

How does "demo mode" help? It helps because we can use the timestamps of the interim results to know what was running when. Bundle-up the interim results across the different instances of netperf for a given point in time and then if one or another of the netperfs weren't running there, we won't be getting bogus results.   We can trade a dependency on time synchronization for explicit load and measure modes to address the issue of skew. And when running netperfs on just a single system, we don't even need time synchronization as there is just the one clock.

[2] There is a whole history behind that which involves times when a gettimeofday() wasn't overly cheap, especially on a multi-CPU system. That discussion is best left for another day. However, even in the aggressive mode, netperf will not always hit the interval. It can go long by the length of time netperf remains in a send or receive socket call depending on the test type.

The downside is we have much more data to process, so we cannot just start the netperfs and post-process the results with a one-line awk script. Of course, one benchmarker's downside is another benchmarker's fodder - there can be value to those interim results just besides avoiding the need for explicit benchmark phases. We can get into that later.

How Many Systems Do I Need?

Having decided on a methodology for our aggregate performance measures, we need to decide how many systems we want to use when implementing it.

Let's consider a simple, four-stream unidirectional bulk-transfer test between two, equally-sized systems, "sut" and "lg1":

```
jonesrick@sut:~/netperf-2.7.1$ for i in seq 1 4
> do
> netperf -H lg1 -P 0 &
> done
...
jonesrick@sut:~/netperf-2.7.1$  87380  16384  16384    10.00    3514.45
 87380  16384  16384    10.01    5606.68
 87380  16384  16384    10.01    2206.63
 87380  16384  16384    10.01    3886.70
...
jonesrick@sut:~/netperf-2.7.1$ awk '{sum += $5}END{print "sum",sum}'
 87380  16384  16384    10.00    3514.45
 87380  16384  16384    10.01    5606.68
 87380  16384  16384    10.01    2206.63
 87380  16384  16384    10.01    3886.70
sum 15214.5
```

Data transferred at ~15 Gbit/s. Where was the bottleneck?

From the netperf results alone, we don't really know. Perhaps "lg1" wasn't able to receive data any faster. Perhaps "sut" wasn't able to send data any faster. Without other information, we really don't know.

Suppose instead of running four streams to one load generator, we ran one stream to each of four load generators, each of which are still equally-sized:

```
jonesrick@sut:~/netperf-2.7.1$ for i in lg1 lg2 lg3 lg4
do
  netperf -H ${i} -P 0 &
done
...
jonesrick@sut:~/netperf-2.7.1$  87380  16384  16384    10.00    2917.23
 87380  16384  16384    10.00    5383.15
 87380  16384  16384    10.01    3633.49
 87380  16384  16384    10.01    3608.63
...
jonesrick@sut:~/netperf-2.7.1$ awk '{sum += $5}END{print "sum",sum}'
 87380  16384  16384    10.00    2917.23
 87380  16384  16384    10.00    5383.15
 87380  16384  16384    10.01    3633.49
 87380  16384  16384    10.01    3608.63
sum 15542.5
```

Now we know something more. We know the bottleneck is in how much "sut" can send. Why? Because we know lg1 can receive at 15 Gbit/s, and can assume[3] the same of lg2, lg3, and lg4, and because we still didn't go any faster than ~15 Gbit/s.

To reinforce that sending and receiving bottlenecks/limits are not always the same, let's flip the direction of data flow with our four load-generator test:

```
jonesrick@sut:~/netperf-2.7.1$ for i in lg1 lg2 lg3 lg4
do
  netperf -t TCP_MAERTS -H ${i} -P 0 &
done
...
jonesrick@sut:~/netperf-2.7.1$  87380  16384  16384    10.00    11326.09
 87380  16384  16384    10.00    7582.45
 87380  16384  16384    10.00    11307.62
 87380  65536  65536    10.00    1803.26
...
jonesrick@sut:~/netperf-2.7.1$ awk '{sum += $5}END{print "sum",sum}'
 87380  16384  16384    10.00    11326.09
 87380  16384  16384    10.00    7582.45
 87380  16384  16384    10.00    11307.62
```

[3] OK, strictly speaking we are ass-u-me-ing, but it is a very reasonable assumption.

```
 87380  65536  65536    10.00    1803.26
sum 32019.4
```

Sut received at ~32 Gbit/s but sent at no more than about 15. One thing I haven't said yet is that sut and the lg instances are all 8-vCPU instances in GCP and so have a network *egress*[4] throttle of 16 Gbit/s.

So, for aggregate tests, we want to have load-generating capacity well in excess of our system under test. That means at least two load-generators of equal "capacity" to the SUT[5], preferably four. Had we used just two load generators in the "sut-receives" test, we might have been limited by their network egress throttles and not seen "sut" receive at >32 Gbit/s.

How Many Streams Do I Need?

It Depends. (™)

OK, that isn't a terribly satisfying answer, but it is accurate. The runemomniaggdemo.sh script is set to run from one to 2*NumvCPUs streams. It has been my experience that this is usually sufficient. However, if you think a true peak hasn't been reached with that many concurrent streams, you can always edit the script.

Why Burst-Mode UDP_RR? My Application Uses TCP.

Netperf is a simple benchmark. Each instance of netperf is pushing data on just a single stream/connection/flow. One could run lots of individual TCP_RR tests, but then to get enough transactions in flight to get the system to saturation would require so many parallel netperfs running (hundreds or thousands) that we *could* be running a context switching benchmark as much as a networking benchmark.

Burst mode is a feature in netperf where it can have more than a single transaction in flight at one time on a single flow/connection (hence the --enable-burst in the short version of the instructions). And in fact, many years ago, the runemomniaggdemo.sh script would use burst-mode and TCP. It was possible to be careful enough about building-up the number of

[4] While Google Cloud does not impose Internal IP *ingress* caps at the present time, that can change in the future.
[5] System Under Test - sometimes in Cloud we will call it the Instance Under Test or IUT

in-flight transactions to “ensure” that each TCP segment sent carried only a single request or a single response depending on direction. Thus we could convert the transaction rate to a packet per second rate when using small request/response sizes. Packets sent per second would equal transaction rate. Packets received per second would equal transaction rate. Simple.

However, even then if there was non-trivial packet loss, things could “bunch-up” and a single TCP segment could end-up carrying multiple requests or multiple responses, breaking our ability to convert transaction rate to a packet rate. And TCP stacks have become much more adept at coalescing user sends into single segments in the name of efficiency and queue minimization and the like, so we cannot make that simple conversion any longer even without packet loss. For example, here is a packet capture from the midst of a burst-mode TCP_RR test with single-byte requests and responses and a burst size of 32, with a 4.15 Linux kernel from Ubuntu 18.04:

```
. . .
19:13:30.384027 IP 10.150.0.2.65432 > 10.150.0.4.65432: Flags [P.], seq 1535:1545,
ack 1559, win 3, options [nop,nop,TS val 2118079153 ecr 4035274653], length 10
19:13:30.384029 IP 10.150.0.4.65432 > 10.150.0.2.65432: Flags [P.], seq 1559:1568,
ack 1545, win 222, options [nop,nop,TS val 4035274653 ecr 2118079153], length 9
. . .
```

You can see from the data lengths that a single TCP segment was carrying either multiple requests or multiple responses. Great when we want to maximize efficiency, not so much when we want simple conversion from transaction rate to packet rate.

Thankfully, UDP is not too smart for our own good here :) No matter the burst size, each transaction with UDP_RR will be the exchange of a UDP datagram carrying a single request and a UDP datagram carrying a single response. And if we keep our request/response sizes small, the IP datagram carrying the UDP datagram carrying our request/response will not be fragmented into multiple IP datagram fragments. So our relationship between transactions per second and packets per second in each direction holds.

Of course, you can still tweak the script to use TCP rather than UDP for the DO_RRAGG testing, but then you must not assume that the transaction/s rate relates cleanly to the packet per second rate. Unless you use a burst size of zero. And then you will likely need many more parallel streams.

Picking the Burst Size

That is done by the “find_max_burst.sh” script. Basically, all it does is keep increasing the burst size (simultaneous transactions in flight) until it no longer sees an increase in transactions per second. For small-packet request/response, this is usually reached once a CPU is saturated on either one side or the other. It then reports the burst size for which it saw peak transactions per second.

The script doesn’t try to zero-in on “the best” burst size. It assumes “good enough” is, well, good enough. Particularly since we will be going to twice as many streams as there are CPUs, we can be reasonably confident that if CPU “oomph” in the SUT is going to be the bottleneck, we will reach it. Still, best to trust but verify from time to time.[6]

If the idea of a different burst size each time bothers you from a standpoint of test reproducibility, feel free to pick a fixed burst size. Say 64. Then the burst size will be the same each time, but you may not have CPU saturation on a single-stream.

Picking the Tests

The runemomniaggdemo.sh script can be set to run several different tests based on “DO_mumble” variables within it:

1. DO_STREAM - run netperf TCP_STREAM tests from the SUT sending data to the load generators.
2. DO_MAERTS - run netperf TCP_MAERTS (MAERTS is STREAM spelled backwards) tests from the SUT receiving data from the load generators.
3. DO_BIDIR - run large request/response, burst-mode TCP_RR tests between the SUT and the load generators in a way which becomes bidirectional bulk transfer on each connection.
4. DO_RRAGG - run single-byte, burst-mode UDP_RR tests between the SUT and the load generators. This is the one we want in the context of this write-up.
5. DO_RR - run a single-byte, synchronous TCP_RR test between the SUT and each load generator in turn.
6. DO_ANCILLARY - gather various bits of information about the SUT.

When the given variable is set to 1 (one) then that test will be run by the script. Otherwise, that test is ignored. DO_ANCILLARY isn’t all that useful. If you enable tests 1 through 5 you will have

[6] If what the system sees as the “NIC” or the network is particularly poor in packet per second performance, that may bottleneck before a or the CPUs of the SUT. And if you increase the request and/or response size, or run bulk-transfer instead, you can end-up at a bitrate limitation before saturating CPUs.

results which can be post-processed for aggregate outbound, inbound, bidirectional and packets-per-second results, along with single-stream latency. This represents a quite reasonable snapshot of the network performance of the SUT for the conditions of the tests.

There is also a "DURATION" variable in the script which controls how long the script will pause at a given number of parallel streams. DURATION=60 is generally good, but you can go longer if you wish. All a matter of what you want to see and how soon. If DURATION multiplied by the number of intervals (times it takes to double one's way to 2x the SUT CPU count) is going to approach 7200 seconds (two hours) you will need to alter the "LENGTH" variable accordingly to ensure netperfs to not terminate prematurely[7]. The DO_RR test script code currently enforces a minimum DURATION of 60 seconds.

Specifying the Load Generators

The runemomniaggdemo.sh script will obtain a list of load generators to use from a file called "remote_hosts" which should have the following format:

```
REMOTE_HOSTS[0]=127.0.0.1
REMOTE_HOSTS[1]=127.0.0.1
NUM_REMOTE_HOSTS=2
```

The first N lines of the remote_hosts file will list the means by which netperf will connect to the load generators - it can be anything you want which will be resolved by getaddrinfo() on the SUT, so it can be a name or an IP address. The last line should be the number of REMOTE_HOSTS entries in the file.[8]
The script will simply round-robin between the entries. If for some reason you wanted more going to one load generator than another, you could simply list that load generator more than once.
The runemomniaggdemo.sh script assumes the remote_hosts file is in the current working directory.

Running the Tests

When looking to run DO_RRAGG tests, the script will attempt to run "find_max_burst.sh" and so the location of that script needs to be in $PATH. If it is unable to run the test, the burst size will

[7] 99 times out of 10 (sic) two hours has been more than enough, and the script will actually terminate (or at least try to) the netperfs when it is finished. It is possible to have netperf run forever by giving the test length of '0' but then if the cleanup ever failed, there would be netperfs forever…
[8] Well, it doesn't *have* to be the last line. The script will simply "source" the file so they can be in any order, but the given ordering is easiest for us humans to understand.

likely be undefined, so be certain it can run the test or hardcode the BURST size in the runemomniaggdemo.sh script. The script does try to pick something else when the find_max_burst.sh explicitly fails but doesn't handle the case when it isn't run.

Assuming you have netperf configured and installed correctly on the SUT and load generators and have a valid remote_hosts file and such you can just do:

```
./runemomniaggdemo.sh
```

And wait. The find_max_burst.sh script does not emit anything while it is running, so there will be a pause before you see any output (this will not be the case with the other sorts of tests the script can be asked to run). After that, things should look something like:

```
jonesrick@sut:~/netperf-2.7.1/doc/examples$ ./runemomniaggdemo.sh
Starting netperfs at 1543515682.686378423 for tps
Starting netperfs on localhost targeting lg1 for tps
Pausing for 60 seconds at 1543515683.697678277 with 1 netperfs running for tps
Resuming at 1543515743.706375807 for tps
Starting netperfs on localhost targeting lg2 for tps
Pausing for 60 seconds at 1543515744.718770555 with 2 netperfs running for tps
Resuming at 1543515804.725203487 for tps
Starting netperfs on localhost targeting lg3 for tps
Starting netperfs on localhost targeting lg4 for tps
Pausing for 60 seconds at 1543515806.742176064 with 4 netperfs running for tps
Resuming at 1543515866.748907082 for tps
Starting netperfs on localhost targeting lg1 for tps
Starting netperfs on localhost targeting lg2 for tps
Starting netperfs on localhost targeting lg3 for tps
Starting netperfs on localhost targeting lg4 for tps
Pausing for 60 seconds at 1543515870.807903819 with 8 netperfs running for tps
Resuming at 1543515930.818608169 for tps
Starting netperfs on localhost targeting lg1 for tps
Starting netperfs on localhost targeting lg2 for tps
Starting netperfs on localhost targeting lg3 for tps
Starting netperfs on localhost targeting lg4 for tps
Starting netperfs on localhost targeting lg1 for tps
Starting netperfs on localhost targeting lg2 for tps
Starting netperfs on localhost targeting lg3 for tps
Starting netperfs on localhost targeting lg4 for tps
Netperfs started by 1543515939.171480464 for tps
Netperfs stopping 1543516002.219344070 for tps
Netperfs stopped 1543516002.255764921 for tps
```

You will see output like that for each test enabled in the script.

Post-processing the Test Results

One your test(s) has finished, you will have a possibly large number of files. They will be of two types:

```
netperf_<testtype>.log
```

Where testtype will be:

- “tps” - Transactions Per Second - the DO_RRAGG test - "netperf_tps.log"
- “bidirectional” - the DO_BIDIR test - "netperf_bidirectional.log"
- “outbound” - the DO_STREAM test - “netperf_outbound.log”
- “inbound” - the DO_MAERTS test - “netperf_inbound.log”
- “sync” - single-stream synchronous TCP_RR - the DO_RR test - “netperf_sync_tps.log”

which will have contents matching what the script emitted to the screen during the run and:

netperf_<testtype>_<instancenum>_to_<loadgenerator>.out

which will have the results of each individual netperf. Testtype is as for the log file. Instance number marks the netperfs in the order in which they were started, starting from “00000”. Loadgenerator will be whatever was in the remote_hosts entry used to pick the load generator for that instance of netperf. These results files will have contents along the lines of:

```
$ less netperf_tps_00000_to_lg1.out
205738.23,Trans/s,0.504,1543515683.205
206284.00,Trans/s,0.500,1543515683.705
208416.49,Trans/s,0.500,1543515684.205
217180.43,Trans/s,0.501,1543515684.706
209459.24,Trans/s,0.518,1543515685.224
…
8039.67,Trans/s,0.531,1543516001.603
67503.78,Trans/s,0.504,1543516002.107
73930.69,Trans/s,0.139,1543516002.246
<other stuff we’ll ignore for the moment>
```

You can see how the *.out files are not all the same size:

```
$ ls -l *.log *.out
```

```
-rw-rw-r-- 1 jonesrick jonesrick  1520 Nov 29 18:42 netperf_tps.log
-rw-rw-r-- 1 jonesrick jonesrick 25275 Nov 29 18:42 netperf_tps_00000_to_lg1.out
-rw-rw-r-- 1 jonesrick jonesrick 21087 Nov 29 18:42 netperf_tps_00001_to_lg2.out
-rw-rw-r-- 1 jonesrick jonesrick 15952 Nov 29 18:42 netperf_tps_00002_to_lg3.out
-rw-rw-r-- 1 jonesrick jonesrick 16782 Nov 29 18:42 netperf_tps_00003_to_lg4.out
-rw-rw-r-- 1 jonesrick jonesrick 11311 Nov 29 18:42 netperf_tps_00004_to_lg1.out
-rw-rw-r-- 1 jonesrick jonesrick 12030 Nov 29 18:42 netperf_tps_00005_to_lg2.out
-rw-rw-r-- 1 jonesrick jonesrick 11223 Nov 29 18:42 netperf_tps_00006_to_lg3.out
-rw-rw-r-- 1 jonesrick jonesrick 11909 Nov 29 18:42 netperf_tps_00007_to_lg4.out
-rw-rw-r-- 1 jonesrick jonesrick  6986 Nov 29 18:42 netperf_tps_00008_to_lg1.out
-rw-rw-r-- 1 jonesrick jonesrick  7058 Nov 29 18:42 netperf_tps_00009_to_lg2.out
-rw-rw-r-- 1 jonesrick jonesrick  6726 Nov 29 18:42 netperf_tps_00010_to_lg3.out
-rw-rw-r-- 1 jonesrick jonesrick  7058 Nov 29 18:42 netperf_tps_00011_to_lg4.out
-rw-rw-r-- 1 jonesrick jonesrick  6876 Nov 29 18:42 netperf_tps_00012_to_lg1.out
-rw-rw-r-- 1 jonesrick jonesrick  6740 Nov 29 18:42 netperf_tps_00013_to_lg2.out
-rw-rw-r-- 1 jonesrick jonesrick  6719 Nov 29 18:42 netperf_tps_00014_to_lg3.out
-rw-rw-r-- 1 jonesrick jonesrick  6634 Nov 29 18:42 netperf_tps_00015_to_lg4.out
```

This stems from how the runemomniaggdemo.sh script works.   It starts with a single netperf/stream/flow, then adds more as it goes. So, the first stream runs the entire length of the test, the second from its start until the end of the test, and so on.   This has properties we may get into later.

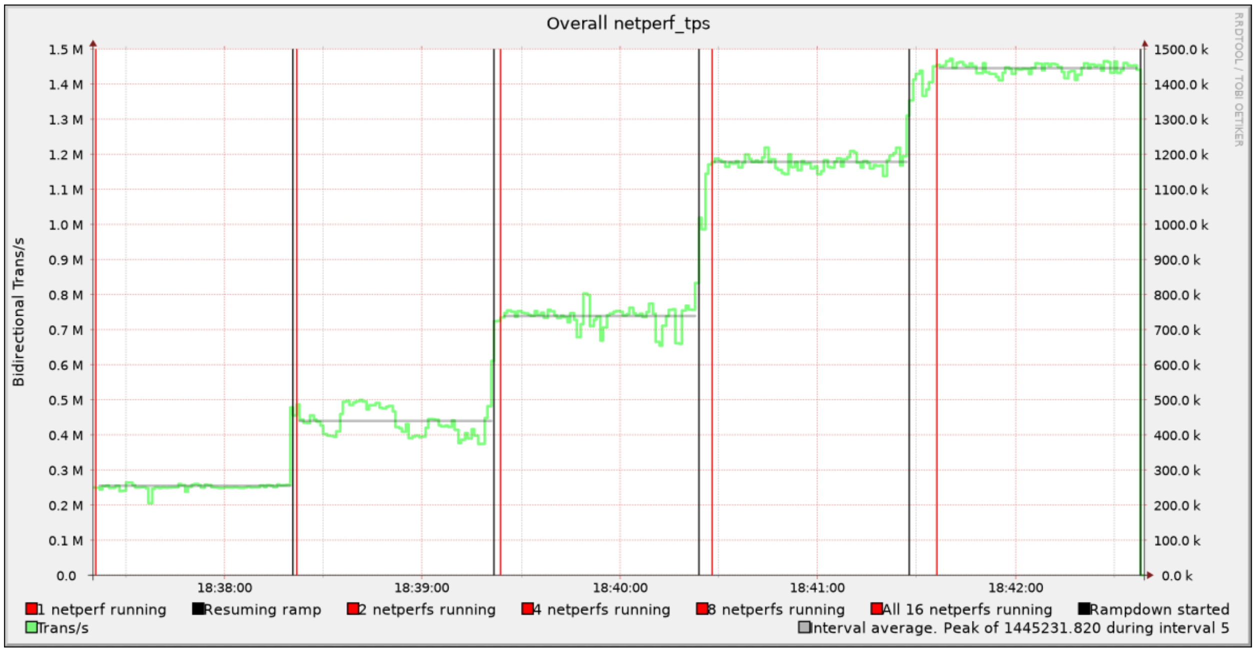
Finally... let's post-process:

```
$ ./post_proc.py --intervals netperf_tps.log
Prefix is netperf_tps
Average of peak interval is 1445231.820 Trans/s from 1543516897 to 1543516957
Minimum of peak interval is 1410508.670 Trans/s from 1543516897 to 1543516957
Maximum of peak interval is 1472099.010 Trans/s from 1543516897 to 1543516957
Average of interval 0 is 254809.880 Trans/s from 1543516642 to 1543516700
Minimum of interval 0 is 205000.290 Trans/s from 1543516642 to 1543516700
Maximum of interval 0 is 478192.120 Trans/s from 1543516642 to 1543516700
Average of interval 1 is 439432.490 Trans/s from 1543516703 to 1543516761
Minimum of interval 1 is 373932.010 Trans/s from 1543516703 to 1543516761
Maximum of interval 1 is 611069.570 Trans/s from 1543516703 to 1543516761
Average of interval 2 is 738828.780 Trans/s from 1543516765 to 1543516823
Minimum of interval 2 is 654077.820 Trans/s from 1543516765 to 1543516823
Maximum of interval 2 is 832886.660 Trans/s from 1543516765 to 1543516823
Average of interval 3 is 1178252.900 Trans/s from 1543516829 to 1543516887
Minimum of interval 3 is 1137790.040 Trans/s from 1543516829 to 1543516887
Maximum of interval 3 is 1309966.880 Trans/s from 1543516829 to 1543516887
Average of interval 4 is 1445231.820 Trans/s from 1543516897 to 1543516957
Minimum of interval 4 is 1410508.670 Trans/s from 1543516897 to 1543516957
Maximum of interval 4 is 1472099.010 Trans/s from 1543516897 to 1543516957
```

We can see the Transaction per second rate for each of the intervals of the test from interval 0 where 2^0 or 1 (one) stream was running through interval 4 where 2^4 or 16 streams were running.[9] The script will default to reporting only the peak interval, the --intervals option has it include all of them in its ASCII output.

Since we know this was an aggregate, burst-mode UDP_RR test, and the transaction rate was ~1.4 million transactions per second, we know the SUT here was sending ~1.4 million packets and receiving ~1.4 million packets per second.

As the script uses rrdtool, it can also generate some graphs. In particular, it will always generate an “overall” graph with a name of the form:
netperf_<testtype>_overall.svg
which you can look at by various means. The one from the test results we see above looks like:



The text may be a bit difficult to read, but the red lines mark the start of an interval. The black lines mark the start of a period where the number of netperfs is ramped-up. Those are getting longer each time because the script pauses one second between each netperf launch[10]. The green line is the second-by-second aggregate throughput, and the grey line is the average for the various intervals. Rrdtool’s facilities are used to get the min, average and max over a measurement interval, which today is defined as starting and ending one second “in” on either

[9] If the SUT had something other than a power of two vCPUs, the number of streams at the last interval will not be a power of two, but 2xNUMvCPUs.
[10] That very likely could be much shorter and still be OK.

side. We keep all the results, even those from when netperfs were being added, which can be occasionally interesting.

Other post_proc.py Options

You have some control over what is listed in the charts, and also whether charts are created for individual netperfs/flows:

```
jonesrick@sut:~/netperf-2.7.1/doc/examples$ ./post_proc.py --help
usage: post_proc.py [-h] [-i] [-I] [-a ANNOTATION] [-t TITLE] filename
positional arguments:
  filename
optional arguments:
  -h, --help            show this help message and exit
  -i, --individual      Generate graphs of individual tests
  -I, --intervals       Emit the results for all intervals, not just peak
  -a ANNOTATION, --annotation ANNOTATION
                        Annotation to add to chart titles
  -t TITLE, --title TITLE
                        String to use for chart title. Default based on test
```

Other Matters

Earlier, we looked at the output of an individual test, and there was some output we ignored. Let’s take a look at that here:

```
$ tail -20 netperf_tps_00000_to_lg1.out
54895.44,Trans/s,0.924,1543516956.323
73910.55,Trans/s,0.558,1543516956.881
55443.87,Trans/s,0.667,1543516957.548
97811.82,Trans/s,0.506,1543516958.054
86643.33,Trans/s,0.564,1543516958.618
113881.19,Trans/s,0.504,1543516959.122
87068.96,Trans/s,0.591,1543516959.713
Stream,UDP,Send|Recv,319.54,166423.74,Trans/s,-1,212992,212992,-1,212992,212992,-1,212992,212992,-1,212
992,212992,0,0,1,1,1,1,54.63,5.59,32.11,0.00,0.00,16.94,S,26.261,12.27,1.05,8.15,0.00,0.00,3.06,S,5.896
,usec/Tran,0,-1.000000,1,-1.000,-1.000,-1.000,166423.736,390.569,64,-1,-1,-1,1.33,1.33,1.33,1.33,-1,8,9
8.62,7,Deprecated,0,-1,8,67.84,7,Deprecated,0,0,0.0.0.0,0,0,lg1,2,53179690,53179626,1.00,1.00,53179626,
53179626,106359252,0,0,8,8,1,1,-1,-1,-1,0,0,53179690,53179690,1.00,1.00,53179690,53179690,106359380,0,0
,8,8,1,1,-1,-1,-1,0,0,Deprecated,Deprecated,Deprecated,Deprecated,Deprecated,Deprecated,Deprecated,Depr
ecated,Deprecated,Deprecated,Deprecated,0xfffffffe,0xfffffffe,0xfffffffe,0xfffffffe,Deprecated,Deprecat
ed,Deprecated,Deprecated,Deprecated,Deprecated,0xfffffffe,0xfffffffe,0xfffffffe,0xfffffffe,Deprecated,D
```

```
eprecated,Deprecated,Deprecated,Deprecated,0,0,0,0,0,Deprecated,0,Deprecated,Deprecated,0,Deprecated,0,
Deprecated,Deprecated,,
a1a69991-807f-40d1-b4a6-bbd0dee40a33,156,52766,293,538,1583,390.40,645.31,-1,-1,0xffffffff,0xffffffff,,
,,,"netperf -H lg1 -D 0.5 -c -C -f x -P 0 -t omni -l 7200 -v 2 -- -r 1 -b 64 -e 1 -T udp -u
a1a69991-807f-40d1-b4a6-bbd0dee40a33 -o all"
Histogram of request/response times
UNIT_USEC     :    0:    0:    0:    0:    0:    0:    0:    0:    0:    0
TEN_USEC      :    0:    0:    0:    0:    0:    0:    0:    0:    0:    0
HUNDRED_USEC  :    0: 444513: 27198880: 12143194: 6692740: 2980848: 1488286: 700252: 324291: 202127
UNIT_MSEC     :    0: 602280: 122579: 52956: 52096: 33147: 23132: 17678: 25286: 16995
TEN_MSEC      :    0: 51750: 6187:  358:   40:   11:    0:    0:    0:    0
HUNDRED_MSEC  :    0:    0:    0:    0:    0:    0:    0:    0:    0:    0
UNIT_SEC      :    0:    0:    0:    0:    0:    0:    0:    0:    0:    0
TEN_SEC       :    0:    0:    0:    0:    0:    0:    0:    0:    0:    0
>100_SECS: 0
HIST_TOTAL:      53179626
```

The very long line starting “Stream,UDP,Send|Recv” is the result of a test-specific (after the “--”) “-o all” in the command line as an output selector. You can see that also causes netperf to emit the command line used to invoke it :) Sadly, since “-P 0” is used to avoid emitting test banners, we don’t get the “headers” saying what each of those are, so doing interesting things with them can be a bit of a challenge. Using keyval output format (“-k all”) in the netperf command-line would result in their being emitted one to a line with their omni selector name. Whether the post-processing script would handle that is unknown. It has code for it but that code hasn’t been well exercised.

The next bit comes thanks to netperf having been ./configure’d with “--enable-histogram” and a “-v 2” on the command line. It is a histogram of all the individual transactions’ round-trip times.

You need to be careful with these!

Remember how the runemomniaggdemo.sh script works by ramping-up the number of connections/flows. That means save for the last M flows (those started in the last interval, though strictly speaking even those weren’t all started at the very same time) those statistics include times with different numbers of flows. That can be interesting if one wants to look at the effect of adding flows on existing ones, say from the first stream’s start using its interim/demo results, since it is one which will be running the entire time (or expected to at least).

If the last interval has a “reasonable” number of added flows, you might be able to wave your hands and say that the histograms and full results from those added flows are a good sampling of what all streams were seeing at that time.

For the “inbound” (DO_MAERTS) tests, the histograms will be time spent in a receive call.
For the “outbound” (DO_STREAM) tests, the histograms will be time spent in send calls. For the _RR tests the histograms will be individual transaction times.

Remember that netperf uses blocking socket calls. Also remember that a send call will complete once the data is in the send socket buffer, **not** when the data is received by the remote.

The UDP_RR test has very crude recovery from lost datagrams. It is based on a socket read timeout on the netperf side, set via a test-specific -e option. The runemomniaggdemo.sh script sets that to one (1) second. If that timeout is hit, netperf will zero-out the histogram of transaction times and issue a new request to get traffic flowing again.

In a classic, physical world, 99 times out of 10 (sic) to a NIC a packet is just a packet. To the NIC a frame with any other payload is just as sweet. And if one managed to achieve N packets per second through a NIC with aggregate, burst-mode UDP_RR, you would be reasonably confident to see that many packets per second with TCP. However, in a software-defined networking world, a packet is not always just a packet to a vNIC (and what is behind it). What that software does/needs to do can be different if the packet is a UDP datagram or a TCP SYN(chronize) segment. So, an aggregate, burst-mode UDP_RR packet per second level through a vNIC may not match what one might expect to see with say a small-URL, non-persistent HTTP connection workload.

Recommended Patches

Since the initial writing of this document, it has been noticed that a UDP_RR test will throw-away all the good work (transactions) completed in a demo interval which includes a socket read timeout. A patch to address this has been sent to the netperf upstream maintainer(s). Until that patch is incorporated into the mainline source, you can take it from here:

```
diff --git a/src/netlib.c b/src/netlib.c
index 884319a..ec501b5 100644
--- a/src/netlib.c
+++ b/src/netlib.c
@@ -3986,16 +3986,6 @@ void demo_first_timestamp() {
   HIST_timestamp(demo_one_ptr);
 }

-void demo_reset() {
-  if (debug) {
```

```
-    fprintf(where,
-           "Resetting interim results\n");
-    fflush(where);
-  }
-  units_this_tick = 0;
-  demo_first_timestamp();
-}
-
 /* for a _STREAM test, "a" should be lss_size and "b" should be
    rsr_size. for a _MAERTS test, "a" should be lsr_size and "b" should
    be rss_size. raj 2005-04-06 */
@@ -4126,6 +4116,26 @@ void demo_interval_tick(uint32_t units)
   }
jonesrick@jonesrick:~/netperf$ git diff src/netlib.c > /tmp/patch
jonesrick@jonesrick:~/netperf$ cat /tmp/patch
diff --git a/src/netlib.c b/src/netlib.c
index 884319a..ec501b5 100644
--- a/src/netlib.c
+++ b/src/netlib.c
@@ -3986,16 +3986,6 @@ void demo_first_timestamp() {
   HIST_timestamp(demo_one_ptr);
 }

-void demo_reset() {
-  if (debug) {
-    fprintf(where,
-          "Resetting interim results\n");
-    fflush(where);
-  }
-  units_this_tick = 0;
-  demo_first_timestamp();
-}
-
 /* for a _STREAM test, "a" should be lss_size and "b" should be
    rsr_size. for a _MAERTS test, "a" should be lsr_size and "b" should
    be rss_size. raj 2005-04-06 */
@@ -4126,6 +4116,26 @@ void demo_interval_tick(uint32_t units)
   }
 }

+/* called when we have a recv timeout on a socket for a UDP_RR test.  until we
+   decide otherwise, we'll force an interval display whenever this happens,
and
+   will not attempt to compensate for the time spent sitting there waiting for
+   the timeout. */
+void demo_reset() {
```

```
+  double actual_interval = 0.0;
+  if (debug) {
+    fprintf(where,
+         "Resetting interim results\n");
+    fflush(where);
+  }
+  HIST_timestamp(demo_two_ptr);
+  actual_interval = delta_micro(demo_one_ptr,demo_two_ptr);
+  demo_interval_display(actual_interval);
+  units_this_tick = 0;
+  temp_demo_ptr = demo_one_ptr;
+  demo_one_ptr = demo_two_ptr;
+  demo_two_ptr = temp_demo_ptr;
+}
+
 void demo_interval_final() {
   double actual_interval;
```