Dispatch No. 009 · Field Notes

Bandwidth Is Not Throughput

iperf, ttcp, hping3 and pchar on a path with a known answer, and why the window, the loss and the round trip set the number long before the bandwidth does.

M.J. Martin
Outlan.net
October 2026 · Version 1.0.2

ABSTRACT

It's not the size of the pipe that sets your transfer rate, rather it's the window, the loss and the round trip.

"We bought a 1 Gig circuit and the file copy still crawls." If you run a network, you have heard some version of that complaint. The circuit is almost never the problem. The circuit is running at the speed it was sold at. What the user is measuring is throughput, and throughput is what one conversation between two hosts actually achieves across the whole path. Bandwidth is only the ceiling.

In The Packet Doesn't Lie we looked at reading a capture byte by byte. This companion is about the tools that put load on a path and measure what comes back: ttcp and its descendants, iperf in both of its current forms, hping3, and pchar. Each one answers a different question. Used on the wrong question, each one gives you a clean, convincing and wrong number.

1. A LAB WITH A KNOWN ANSWER

You cannot grade a measurement tool against a path you don't understand. So before running anything, I built one where I knew the answer: a client, a router and a server, each in its own Linux network namespace, joined by two emulated links.

Table 1. The test path

Link	Rate	One-way delay	Queue
Client 192.0.2.1 to router 192.0.2.2	100Mbps	2 ms	150 KB
Router 192.0.2.65 to server 192.0.2.66	20Mbps	20 ms	150 KB

The propagation delay adds up to a 44 ms round trip, and ping measured 45.0 ms once serialization and processing were added. The 20Mbps link is the bottleneck. Every number below should be read against those two facts (there's that 45 again, and it will keep coming back).

> **NOTE** The test host's kernel had no netem queueing discipline, so the links are a small userspace emulator on TUN devices that applies rate, delay, a drop-tail queue and random loss per direction. The router is a real Linux router, so TTL, ICMP and forwarding behave normally. On an ordinary Linux host, tc netem does the same job. Userspace timing adds a little jitter, which matters for exactly one tool in this dispatch, and I will point it out when we get there.

2. TTCP: WHERE THIS STARTED

The idea of measuring TCP from memory to memory goes back to 1984. Terry Slattery at the Naval Academy and Mike Muuss at the Ballistic Research Laboratory (yes, the same Mike Muuss who wrote ping) adapted a vendor's network test program into ttcp, "Test TCP." It was used to compare the Berkeley and BBN TCP stacks, to help DARPA decide which one went into 4.3BSD. The point was to test the network without the disk. FTP numbers were limited by disk I/O, and ttcp generated its buffers in memory instead.

The original syntax is still worth knowing, because you will find ttcp on old routers and in old runbooks. The receiver starts first, then the transmitter:

```
ttcp -r -s                    # receiver: sink the data
ttcp -t -s 192.0.2.66         # transmitter: source test data
```

Muuss also liked that it behaves like a pipe across the wire, so `tar cf - directory | ttcp -t host` on one side and `ttcp -r | tar xvpf -` on the other copies a directory with no file service at all. The original ttcp is not packaged for current Linux distributions, so for the lab I used nuttcp, the maintained descendant (by way of SGI's nttcp). Its data port defaults to 5001, which iperf2 also uses, so I moved it:

```
nuttcp -S                                 # on the server
nuttcp -p5101 -T10 -i2 192.0.2.66         # on the client
   23.6875 MB /  10.71 sec =   18.5584 Mbps 0 %TX 2 %RX 0 retrans 45.11 msRTT
```

18.56Mbps, no retransmissions, a 45.11 ms round trip, and the CPU load on both ends. Note the 10.71 seconds for a 10-second test. The measurement is taken at the receiver, after the last byte arrives. With the ttcp family, the receive-side number is the one to trust, because the sender's writes can finish while data is still in flight.

3. IPERF3: THE WINDOW SETS THE CEILING

iperf3 is what most people reach for today. With defaults, it filled the path:

```
iperf3 -s                                 # on the server
iperf3 -c 192.0.2.66 -t 10 -O 2           # on the client, skip the first 2 seconds
[  5]   0.00-10.05  sec  22.5 MBytes  18.8 Mbits/sec                  receiver
```

18.8Mbps on a 20Mbps link is about what you should expect once TCP and IP headers take their share of each 1,500-byte packet. The `-O 2` omits the first two seconds, so slow start does not drag the average down.

Now limit the window. TCP can only have one window of unacknowledged data in flight, then it has to wait for the acknowledgments to come back. So the most a single connection can carry is the window divided by the round trip:

```
throughput <= window / RTT
iperf3 -c 192.0.2.66 -t 8 -O 2 -w 64K
```

Table 2. Window size against throughput, 45 ms round trip, 20Mbps bottleneck

Window (`-w`)	window / RTT	Measured
16 KB	2.91Mbps	3.26Mbps
32 KB	5.83Mbps	6.51Mbps
64 KB	11.65Mbps	12.6Mbps
128 KB	23.3Mbps (above the link)	19.3Mbps (link-limited)

The measured numbers run about 10 percent above the simple calculation. Linux doubles the buffer size you request and advertises part of it as the window, so the effective window lands a little above the number you typed. The ratio holds, though. A 64 KB window caps a connection at about 12Mbps on a 45 ms path no matter how big the circuit is. Put that same 64 KB window on a 1 Gig circuit and you still get about 12Mbps (now you know where the crawling file copy came from).

The window you need is the bandwidth-delay product: 20Mbps times 45 ms is 112.5 KB. Modern stacks tune their buffers automatically and use window scaling (RFC 7323) to advertise windows above 64 KB, so you mostly see this problem with old software, applications that set their own small buffers, and middleboxes that strip the window scale option.

4. LOSS, AND THE ALGORITHM THAT DECIDES WHAT LOSS MEANS

Next I told the emulator to drop packets at random on the 20Mbps link. I expected the throughput to fall. It didn't. One packet in a hundred lost, and iperf3 still reported 18.9Mbps.

That however was not the path. It was the kernel. The test host defaults to BBR congestion control, and BBR does not treat random loss as a congestion signal the way loss-based algorithms do. Run the same test with CUBIC (the usual Linux default) or classic Reno and the picture changes completely:

```
iperf3 -c 192.0.2.66 -t 20 -O 3 -C cubic
```

Table 3. Random loss against congestion control, one stream, 20 seconds

Loss	CUBIC	Reno	BBR	Mathis estimate
none	19.3Mbps	19.3Mbps	18.8Mbps	link-limited
0.1 percent	9.36Mbps	9.83Mbps	18.7Mbps	9.93Mbps
1 percent	2.51Mbps	2.77Mbps	18.9Mbps	3.14Mbps

The last column is the Mathis formula from 1997, the rule of thumb for loss-based TCP: throughput is roughly the segment size divided by the round trip, times 1.22 over the square root of the loss rate. With a 1,448-byte segment and a 45 ms round trip, it predicted 9.93Mbps at 0.1 percent loss, and Reno delivered 9.83. One lost packet in a thousand cut the throughput in half. One in a hundred cut it to about a seventh.

So before comparing two throughput numbers, ask what congestion control each end was running. The same path gave 2.51Mbps and 18.9Mbps in the same session, and the only thing that changed was one sysctl. If the numbers do not make sense, check `net.ipv4.tcp_congestion_control` on both hosts before you open a ticket with the carrier.

5. PARALLEL STREAMS HIDE THE PROBLEM

The usual advice when a single stream is slow is to add more streams. It works, which is exactly the trouble:

```
iperf3 -c 192.0.2.66 -t 20 -O 3 -C cubic -P 4
```

Table 4. Parallel CUBIC streams at 1 percent loss

Streams (`-P`)	Total measured
1	2.93Mbps
4	9.99Mbps
8	18.9Mbps

Eight streams filled the link through 1 percent loss. (The single stream came in at 2.93 here and 2.51 in Table 3, which is the normal run-to-run spread at this loss rate.) A test with `-P 8` tells you the path can carry 19Mbps in aggregate. It does not tell you what your backup job or file copy will see, because that application opens one connection. Test the way the application behaves, then decide whether to fix the loss or change the application.

6. UDP: OFFERED LOAD AND WHAT ARRIVES

UDP has no congestion control, so iperf3 sends at whatever rate you ask for, and the receiver reports what arrived:

```
iperf3 -c 192.0.2.66 -u -b 15M -t 10
[  5]   0.00-10.05  sec  17.9 MBytes  14.9 Mbits/sec  0.313 ms  0/12948 (0%)  receiver
iperf3 -c 192.0.2.66 -u -b 30M -t 10
[  5]   0.00-10.10  sec  23.5 MBytes  19.5 Mbits/sec  0.477 ms  8858/25893 (34%)
receiver
```

At 15Mbps, nothing was lost and jitter was 0.3 ms. At 30Mbps, 19.5Mbps arrived and 34 percent was dropped at the bottleneck queue. That makes UDP the most direct way to find a bottleneck rate, and also the most reliable way to break someone else's traffic while you do it. On a production path, start low and step up.

7. IPERF2: COUNTING TRANSACTIONS

A reasonable question at this point is why anyone still runs iperf2. The two tools split around 2010. ESnet started iperf3 as a clean rewrite (single-threaded until version 3.16 in 2023), and iperf2 was revived in 2014 by Bob McMahon at Broadcom, with a focus on latency. They do not talk to each other, so the client and server have to match.

The iperf2 feature that matters here is bounceback. The client sends a small request, the server sends it back, and iperf2 counts how many round trips per second it gets. That is a transaction rate, not a bit rate, and it is the number the user actually feels:

```
iperf -s -e                                     # on the server
iperf -c 192.0.2.66 --bounceback -t 10 -e       # 100-byte request and response
[  1] 0.0000-10.0499 sec  19.5 KBytes  15.9 Kbits/sec    100=45.248/44.908/47.408/0.355
ms    0    5K/45656 us    22 rps
```

22 transactions per second. That is 1 divided by the 45 ms round trip (there it is again), and it used 15.9Kbps, less than a tenth of one percent of the link. With 64 KB requests (`-l 64K`) the rate fell to 9.9 per second, because each transaction now also waits on serialization through the 20Mbps link. A chatty application that makes one request at a time will never go faster than this, on any circuit. Buying more bandwidth does not move the number. Shortening the path or cutting the number of round trips does.

This is the same argument I made in my 2011 LISA talk, where I set bandwidth-centric design against session-centric design. What was true then is still true today.

8. HPING3: WHEN PING LIES

hping3 is a packet crafter, written by Salvatore Sanfilippo (antirez, who went on to write Redis). The last release is from 2005, and the major Linux distributions still package it. For measurement, its most useful trick is a "ping" built from TCP.

To show why, I added one rule on the router that drops ICMP echo requests, which is a common enough firewall policy. Ping went to 100 percent loss. The host was fine:

```
hping3 -S -p 5201 -c 5 192.0.2.66
len=44 ip=192.0.2.66 ttl=63 DF id=0 sport=5201 flags=SA seq=0 win=64240 rtt=45.4 ms
...
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 45.3/45.8/46.2 ms
```

A SYN-ACK (`flags=SA`) from an open port gives you a round trip through the same firewall rules the application uses. A closed port answers with RST-ACK (`flags=RA`, try `-p 81`), which still proves the host is up. `--traceroute` walks the path with TCP SYNs instead of UDP or ICMP, which can get through filters that stop the classic traceroute.

hping3 can also show serialization delay by hand. With `-d` setting the payload size, 20 SYNs at each size gave a minimum round trip of 45.3 ms with no data and 45.8 ms with 1,400 bytes. The arithmetic says 1,400 extra bytes should add 0.11 ms on the 100Mbps link and 0.56 ms on the 20Mbps link, 0.67 ms in all in the direction they travel. The measured difference was 0.5 ms. Taking the minimum of many samples, so queueing does not hide the difference, is exactly what pchar automates.

Last, Path MTU. I dropped the router's far-side link to an MTU of 1,400 and sent SYNs with Don't Fragment set (`-y`):

```
hping3 -n -S -p 5201 -y -d 1300 -c 2 192.0.2.66    # 1,340-byte packet: SYN-ACK
hping3 -n -S -p 5201 -y -d 1400 -c 2 192.0.2.66    # 1,440-byte packet:
ICMP Fragmentation Needed/DF set from ip=192.0.2.2
ICMP Fragmentation Needed/DF set from ip=192.0.2.2
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

Read the lines, not the summary. hping3 counted the two ICMP errors as replies and reported "0% packet loss" for packets that never reached the server. The capture told the truth, using the Fragmentation Needed filter from the last dispatch:

```
tcpdump -nn -i eth0 'icmp[0] = 3 and icmp[1] = 4 and icmp[6:2] < 1500'
IP 192.0.2.2 > 192.0.2.1: ICMP 192.0.2.66 unreachable - need to frag (mtu 1400), length
556
```

9. PCHAR: CHARACTERIZING EACH HOP

pchar answers a question the others can't: which hop is the slow one? It is Bruce Mah's open source reimplementation, written at Sandia, of Van Jacobson's pathchar from 1997. At each hop it sends probes of many sizes, keeps the minimum round trip for each size, and fits a line through them. The slope of that line is

the time per byte, which is one over the bandwidth. Subtract one hop's line from the next and you get that hop's own latency and bandwidth.

```
pchar -n -R 16 -I 64 -g 0.05 -l 192.0.2.1 192.0.2.66
```

Table 5. pchar against the configured path

Hop	Configured	pchar	Fit (r^2)
1 (client to router)	100Mbps, 4 ms RTT	145Mbps, 4.56 ms	0.33
2 (router to server)	20Mbps, 40 ms RTT	18.4Mbps, 40.4 ms	0.97
Path	bottleneck 20Mbps	bottleneck 18.4Mbps, pipe 103,291 bytes	

It found the bottleneck, within 8 percent, along with its latency, and it reported the path's pipe size (bandwidth times delay) as 103 KB. Compare that with the 112.5 KB bandwidth-delay product from the window section. It got the fast hop wrong. A 100Mbps link adds only 120 microseconds to a 1,500-byte packet, small enough that timing noise swamps the slope, and the r^2 of 0.33 says so. My userspace links add more jitter than real hardware would, so treat this as a best case for that problem, not a worst case. Mah has acknowledged the other limits: one probe in flight at a time, so parallel links and port channels look like a single link, and switched hops with queues below IP confuse the estimate.

It is also slow. With the defaults (46 packet sizes, 32 repetitions, a quarter-second gap) it sends 1,472 probes per hop, which comes to several minutes per hop on this path. With the options above it took 55 seconds for both hops. pchar has not been developed since version 1.5, and Mah said as much in the README. It still works, and on a path you control it is still the quickest way to see where the serialization delay lives.

10. WHICH TOOL FOR WHICH QUESTION

Table 6. Matching the tool to the question

Question	Tool	Watch out for
What can one TCP connection carry?	iperf3 or nuttcp, one stream	window size, congestion control on both ends
What can the path carry in aggregate?	iperf3 `-P`	it hides loss that one application stream will feel
What is the bottleneck rate and jitter?	iperf3 `-u -b`	UDP will push other traffic out of the way
How many transactions per second?	iperf2 `--bounceback`	iperf2 and iperf3 do not interoperate
Is the host up behind a firewall?	hping3 `-S -p`	the summary line counts ICMP errors as replies
Which hop is slow?	pchar	slow, noisy on fast hops, blind to parallel links

11. KEEP THE BASELINE

Every number in this dispatch came from a path where I knew the answer first. That is the habit worth keeping. Run these tests when nothing is wrong, write down what the path does on a good day (the window, the round trip, the congestion control, the transaction rate), and keep that record with the network documentation. When the call comes in that the network is slow today, you will have yesterday's numbers to compare against, and the argument will take five minutes instead of five days.

The link has a speed. The path has a throughput. Measure the one the user is complaining about.

REFERENCES

[1] M. Mathis, J. Semke, J. Mahdavi and T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm," ACM SIGCOMM Computer Communication Review, July 1997.

[2] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh and V. Jacobson, "BBR: Congestion-Based Congestion Control," ACM Queue, vol. 14, no. 5, 2016.

[3] L. Xu, S. Ha, I. Rhee, V. Goel and L. Eggert, Ed., "CUBIC for Fast and Long-Distance Networks," RFC 9438, August 2023. rfc-editor.org

[4] D. Borman et al., "TCP Extensions for High Performance," RFC 7323, September 2014. rfc-editor.org

[5] J. Mogul and S. Deering, "Path MTU Discovery," RFC 1191, November 1990. rfc-editor.org

[6] M. Muuss, "The Story of the TTCP Program." ftp.arl.mil

[7] T. Slattery, "The Origin of TTCP (Test TCP)," NetCraftsmen. netcraftsmen.com

[8] ESnet, iperf3 FAQ: history and differences between iperf2 and iperf3. software.es.net

[9] S. Sanfilippo, hping3(8) manual page. mankier.com

[10] B. A. Mah, pchar 1.5 README. sources.debian.org

[11] B. A. Mah, "Re: Traceroute versus other performance measurement," NANOG mailing list, November 2000. seclists.org