TCP-NV: An Update to TCP-Vegas

Author: Lawrence Brakmo
nv<at>brakmo.org

Document History

6/22/15	Created
8/26/15	Updates to NV and more variety of experiments now including Reno, Cubic, NV and CDG. Also more complex scenarios, including 10KB vs. 1MB RPCs.

1. Overview

TCP-NV is a major update to TCP-Vegas (NV stands for New Vegas). Like Vegas, NV is a delay based congestion avoidance mechanism for TCP. Its filtering mechanism is similar: it uses the best measurement in a particular period to detect and measure congestion. It develop to coexist with modern networks where links bandwidths are 10 Gbps or higher, where the RTTs can be 10's of microseconds, where interrupt coalescence and TSO/GSO can introduce noise and nonlinear effects, etc.

Although the underlying concept of NV is similar to Vegas, it was developed from its own basic principles. Suppose that we plot rate vs. cwnd for every packet for which an ACK is received, where rate is defined as: cwnd bytes / RTT of packet acked. Note that we use cwnd in this discussion for simplicity, in practice we use in-flight or unacknowledged. Rather than just points, we end up with vertical bars for values of cwnd due to transient congestion or noise in the measurements (see Figure 1).

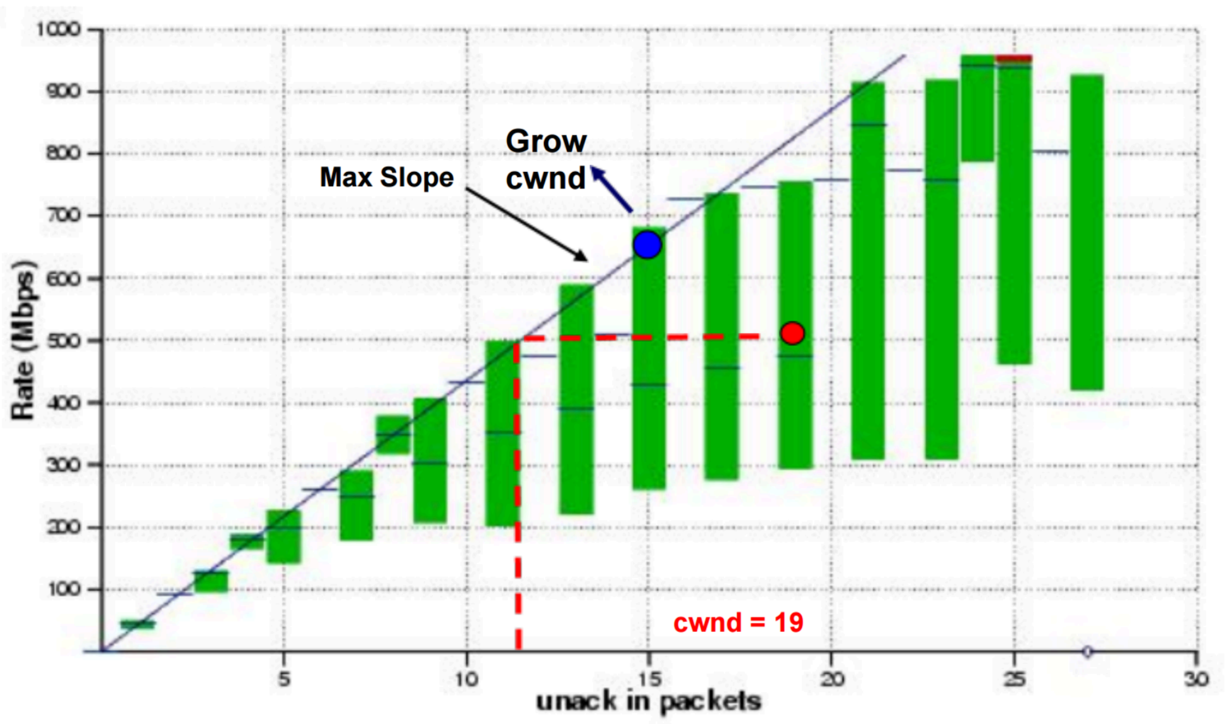


Figure 1: rate vs. cwnd

We use the maximum slope of the top of the bars to indicate the best we have been able to do in the past. In a well tuned system, the top of the bars is bounded by a straight line going through the origin. The idea is that as long as we are not congested, doubling the amount of data we sent per RTT will double the rate. Note that this assumption can be broken by choosing large NIC coalescence parameters. Hence it is important to check that the system behaves linearly with the chosen parameters before enabling NV.

Then, new measurements (rate and cwnd), can either fall close to the boundary line (blue dot) or below (red dot). A measurement above the line will automatically update the line by increasing its slope so the measurement will fall on the new line. If the new measurement is close to the line, then we increase the cwnd. If the measurement is below the line, it means that we have seen equal performance in the past with a lower cwnd (in the example in Figure 1, we achieved similar performance with cwnd of 12), so we will decrease the cwnd. The decrease is done multiplicative, rather than instantaneously, in case the new measurement is noisy.

To filter out bad measurements, we collect many measurements and then use the best one before making a congestion determination. The particular details will be described in the following section.

There is a Linux patch available.

2. Implementation

The current Linux implementation of TCP-NV has many parameters to support experimentation. The majority of the code is located in the tcp_nv.c module file. Only small changes are required in the kernel to store the number of bytes in transit when each packet is sent. This is information is then made available so TCP-NV can use it when the ACKs arrive.

A prior implementations of TCP-NV was based on Cubic, that is, when cwnd growth was allowed it would grow with Cubic dynamics. This required increasing the ca struct space. Since a lot of the mechanisms in Cubic are not needed with congestion avoidance (hystart, varying cwnd growth based on distance to cwnd value when prior losses occurred), the current implementation uses a modified a simpler algorithm for cwnd growth. See definition below of `nv_cwnd_growth_factor` parameter.

The main logic of TCP-NV is as follows:
When a packet is ACKed, we measure the RTT (in microseconds) and either use it as is or compute a (fast) moving average. Although our explanation used a slope to represent the congestion line, the implementation uses the minRTT for simplicity. Theoretically we shouldn't average RTTs since we are interested in the minRTT; however in practice we are not interested in a minRTT which is rare due to the noise in the measurements (as opposed to congestion).

We use the avgRTT and bytes in flight when the packet was sent to calculate the rate. If the current rate is faster than the previous maxRate, we update maxRate. MaxRate is the maximum rate seen since the last congestion determination.
Then we update the current minRTT if necessary, and the future minRTT. In order to deal with changes in the network, we reset the minRTT every so often (`nv_reset_period` seconds). But rather than do a cold resets (i.e. minRTT = minRTT<<2), we do a warm update where we start collecting a future minRTT and then switch to it.

Every RTT we do the core work:

- If we only received one ACK per RTT, then we allow cwnd growth since one ACK per RTT can lead to issues due to TSO/GSO, etc.
- Disabled by default: Every so often we decrease the cwnd temporarily to get RTT measurements that are not inflated due to our own packets. This works best with small number of flows, but increases P99 latency significantly.
- Calculate the appropriate cwnd for the current maxRate and minRTT; set max_win to this plus nv_pad
- If snd_cwnd > max_win (i.e. we see congestion), check that we have enough RTT measurements (`nv_dec_eval_min_calls)` and enough RTT measurement periods (`nv_rtt_min_cnt`). If not, return. Otherwise decrease cwnd, disable cwnd growth and return.
- If `snd_cwnd < max_win - nv_pad_buffer`, enable cwnd growth immediately
- Else, disable cwnd growth (but keep cwnd as is).
- Reset maxRate and other counters.

3. Parameters

The main parameters (with default values) whose values can be changed when the module is loaded, are:

`nv_pad (10)`
Number of extra packets to keep in the network

`nv_pad_buffer (2)`
Don’t grow cwnd if number of extra packets in the network is >= `nv_pad - nv_pad_buffer`

`nv_reset_period (5)`
How often (in seconds) to reset minRTT by switching to a warmed minRTT value

`nv_min_cwnd (10)`
cwnd will not be decreased below this due to congestion avoidance (will decrease due to losses, slow-start-after-idle, etc.)

`nv_dec_eval_min_calls (60)`
We need to collect this many measurements (RTT, rate) before deciding there is congestion. The decision of no congestion can be made immediately (at end of RTT period).

`nv_ssthresh_eval_min_calls (30)`
When in slow-start we make a decision quicker because cwnd grows so fast

`nv_rtt_min_cnt (2)`
We wait this many RTTs before deciding there is congestion. The decision of no-congestion can be made every RTT.

`nv_cong_decrease_mult (38 equivalent to 30%)`
How quickly to decrease cwnd when congestion is detected.
cwnd -= max(2U, ((cwnd - max_win) * `nv_cong_decrease_mult`) >> 7);

`nv_ssthresh_factor (8 equivalent to cwnd)`
How to update ssthresh after congestion is detected. If =8, ssthresh is set to updated cwnd. Larger value => cwnd will quickly increase to that value.
ssthresh = (`nv_ssthresh_factor` * max_win) >> 3;

`nv_rtt_factor (128)`
Used for calculating moving RTT average:
avgRTT = (newRTT * `nv_rtt_factor` + avgRTT * (256 - `nv_rtt_factor`)) >> 8

`nv_rtt_cnt_inc_delta (0 - disabled)`

How long (in RTTs) to decrease cwnd every (averaged) 256 RTTs in order to get a better minRTT measurement. This feature increases tail latency (i.e. will increase latency for those RPCs that occur during the cwnd decrease) so it is disabled by default.

`nv_dec_factor (8 - disabled)`
How much to periodically decrease cwnd by
cwnd = (cwnd * `nv_dec_factor`) >> 3

`nv_loss_dec_factor (512 equivalent to 50%)`
How much to decrease cwnd when losses occur.
cwnd = (cwnd * `nv_loss_dec_factor`) >> 10
This is equivalent to Reno. This works well when network is so congested that NV cannot prevent many losses. Working on making this dynamic based on network conditions.

`nv_cwnd_growth_factor (0 - disabled)`
Controls how quickly to grow cwnd (experimental) when no congestion is detected.
0 => linear growth (like Reno)
1 => cwnd grows by 1, 1, 2, 2, 4, 4, … (i.e. 1st 2 RTTs by 1, next 2 RTTs by 2, …)
2 => cwnd grows by 1, 1, 1, 2, 2, 2, 4, 4, 4, …
Growth reset to linear after congestion is detected
Within a datacenter Reno behavior is best, faster increase is better (i.e. like Cubic) as RTT increases. Looking into making this dynamic based on RTT.

4. Experimental Results

I've currently only test the new version within in a rack. More testing will be done in the future and this document will be updated with the results. The HW RTT between hosts is 20-30us, links are 10Gbps. Receive and send buffers are large (> 4MB). In all cases there is one or more client sending to 1 or 2 receivers using netperf. The transfer type is either back-to-back 10K RPC, 1MB RPC, or STREAM.

Latency for RPC is the time (in us) from writing the request to receiving the reply. For Stream is the time for the fixed size writes to complete.

There is a web page showing these experiments with more detail at: http://www.brakmo.org/networking/tcp-nv/TCP-NV.html. Clicking on the experiment number will show graphs of Goodput, cwnd, RTTs, etc.

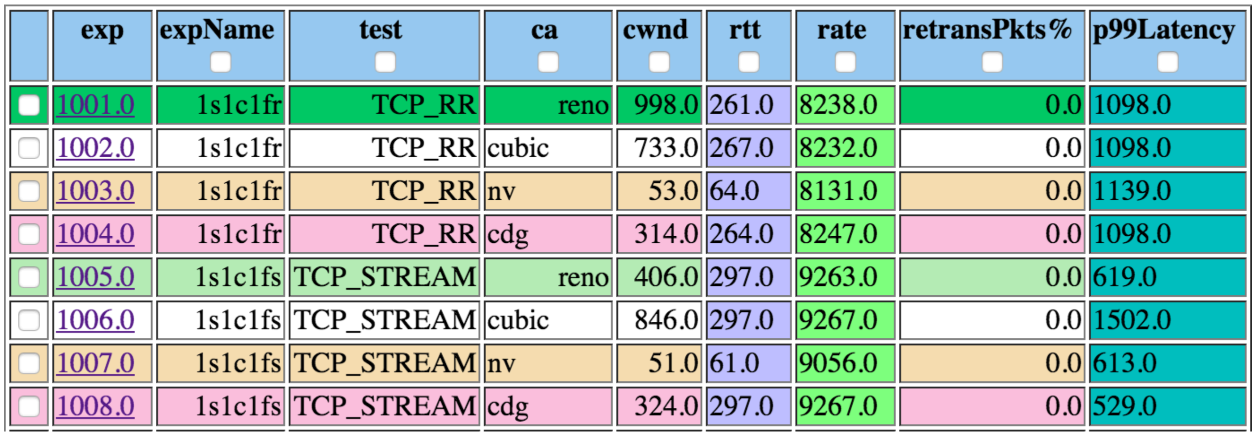
Experiment type notation:
X **s** Y **c** Z **f** [v] [r|s] [p]
Where:
X - number of clients hosts (senders)
Y - number of servers (receivers)
Z - number of flows per client
v - versus. Examples: 10K vs. 1M RPCs, NV vs. Cubic

r - RPC
s - STREAM
p - partial (the 2nd flow is started 15 secs after the 1st one starts, and ends 15 secs before)
Examples:
1s1c1fr - 1 client host sending 1 flow doing 1MB RPCs to 1 server
1s2c1fs - 2 clients hosts sending 1 flow each doing STREAM to 1 server
1s21fvsp - 2 client hosts competing, sending 1 flow each doing STREAM.
Example, the first flow is NV and the second flow cubic, and starts in the middle of the first one and ends before. This shows the unfairness that results when congestion avoidance competes with congestion control

In the experiment Tables, “exp” entries ending in “.0” are a summary of the experiment while those ending with something other than “.0” are individual flow results if they start with “.1” and progress sequentially (“.2”, …) or averaged otherwise.

4.1 1-Client and 1-Flow per Client

In this scenario there is queue buildup at the host, but not at the switch. There can be no losses since all links have the same bandwidth. We see in Table 1 that Reno/Cubic/CDG achieve higher throughput than NV (1-2% higher), but use a cwnd much larger than necessary (314 to 998, where 60 would suffice). As a result the RTT as seen by TCP is much higher for Reno/Cubic/CDG (260us) vs. NV (64us).



	exp ☐	expName ☐	test ☐	ca ☐	cwnd ☐	rtt ☐	rate ☐	retransPkts% ☐	p99Latency ☐
☐	1001.0	1s1c1fr	TCP_RR	reno	998.0	261.0	8238.0	0.0	1098.0
☐	1002.0	1s1c1fr	TCP_RR	cubic	733.0	267.0	8232.0	0.0	1098.0
☐	1003.0	1s1c1fr	TCP_RR	nv	53.0	64.0	8131.0	0.0	1139.0
☐	1004.0	1s1c1fr	TCP_RR	cdg	314.0	264.0	8247.0	0.0	1098.0
☐	1005.0	1s1c1fs	TCP_STREAM	reno	406.0	297.0	9263.0	0.0	619.0
☐	1006.0	1s1c1fs	TCP_STREAM	cubic	846.0	297.0	9267.0	0.0	1502.0
☐	1007.0	1s1c1fs	TCP_STREAM	nv	51.0	61.0	9056.0	0.0	613.0
☐	1008.0	1s1c1fs	TCP_STREAM	cdg	324.0	297.0	9267.0	0.0	529.0

Table 1: One client and one flow per client

Chart 1 shows the rate (goodput) in Gbps and the average cwnd that achieved that rate. The red diamonds correspond to the Y axis on the right (cwnd). NV has 1% lower goodput on the RPC and 2% lower goodput on the Stream experiments. However, NV’s cwnds were much smaller than those of the other RCP variants

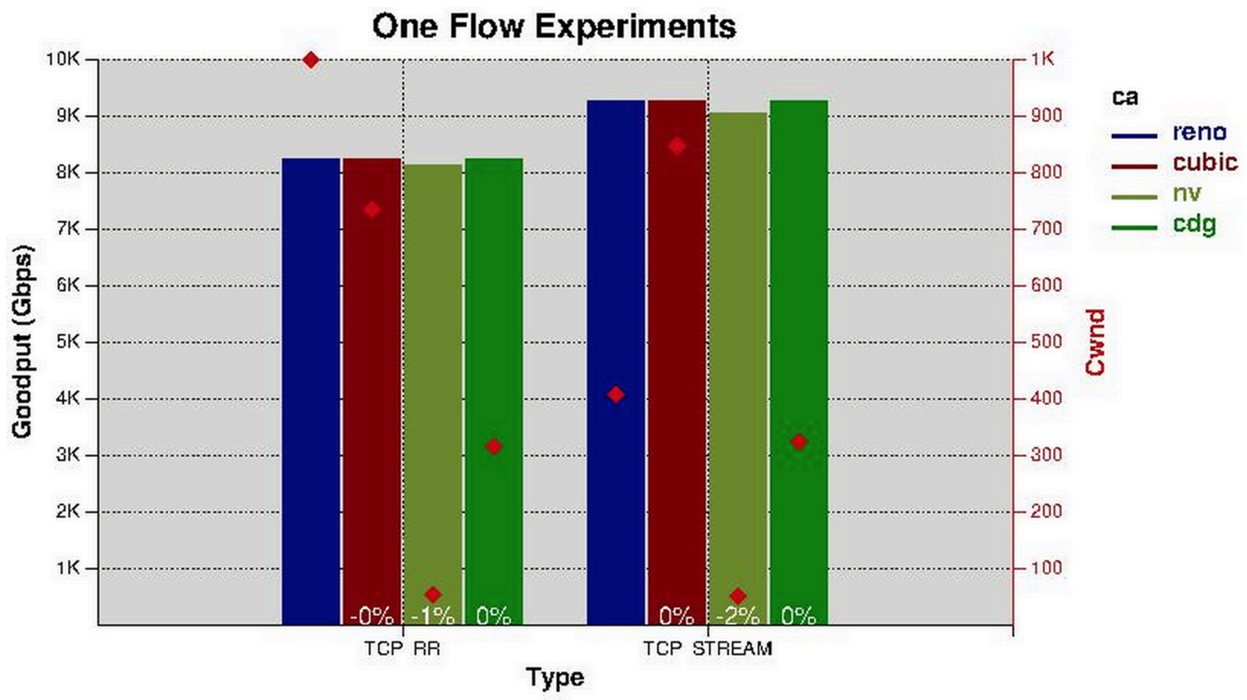
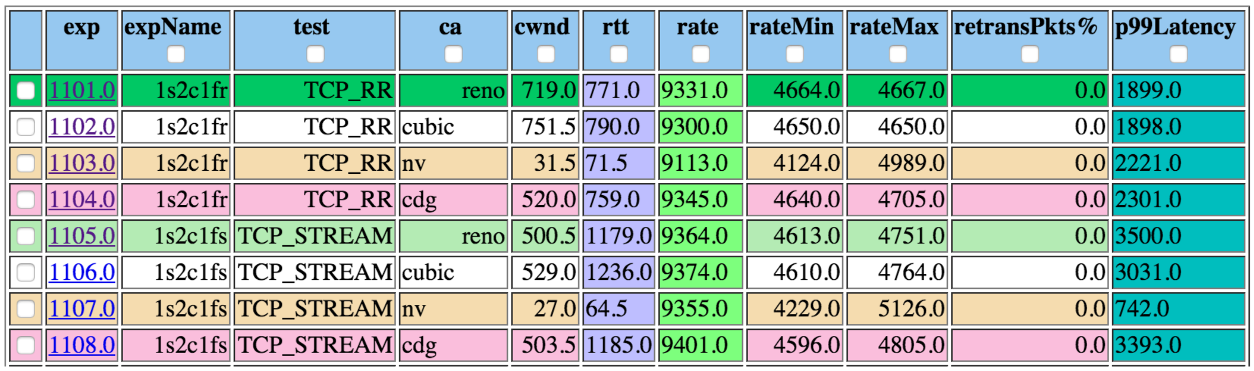


Chart 1: Rate and Cwnd for 1-flow experiments

4.2 2-Clients and 1-Flow per Client

In this scenario there is queue buildup at the switch, specially when streaming where there are a few retransmissions for non-NV transfers (% too low to show in table). As seen in Table 2, Reno/Cubic/CDG still achieves higher aggregate goodput (1-3%). The RTTs are now much higher for non-NV (0.7 - 1.2ms) but not for NV (65-72us). NV is not as fair as seen by the rateMin and rateMax columns (in this experiment they show the rates of the 2 flows).

Figures 2a,b and 3a,b show Goodput and cwnd for each flow for Cubic and NV respectively in the streaming test (1s2c1fs). It shows that Cubic is fair at a larger time scale (~10 secs). The vertical bars in the cwnd graph show when retransmissions occurred.



	exp	expName	test	ca	cwnd	rtt	rate	rateMin	rateMax	retransPkts%	p99Latency
☐	1101.0	1s2c1fr	TCP_RR	reno	719.0	771.0	9331.0	4664.0	4667.0	0.0	1899.0
☐	1102.0	1s2c1fr	TCP_RR	cubic	751.5	790.0	9300.0	4650.0	4650.0	0.0	1898.0
☐	1103.0	1s2c1fr	TCP_RR	nv	31.5	71.5	9113.0	4124.0	4989.0	0.0	2221.0
☐	1104.0	1s2c1fr	TCP_RR	cdg	520.0	759.0	9345.0	4640.0	4705.0	0.0	2301.0
☐	1105.0	1s2c1fs	TCP_STREAM	reno	500.5	1179.0	9364.0	4613.0	4751.0	0.0	3500.0
☐	1106.0	1s2c1fs	TCP_STREAM	cubic	529.0	1236.0	9374.0	4610.0	4764.0	0.0	3031.0
☐	1107.0	1s2c1fs	TCP_STREAM	nv	27.0	64.5	9355.0	4229.0	5126.0	0.0	742.0
☐	1108.0	1s2c1fs	TCP_STREAM	cdg	503.5	1185.0	9401.0	4596.0	4805.0	0.0	3393.0

Table 2: Two clients and one flow per client

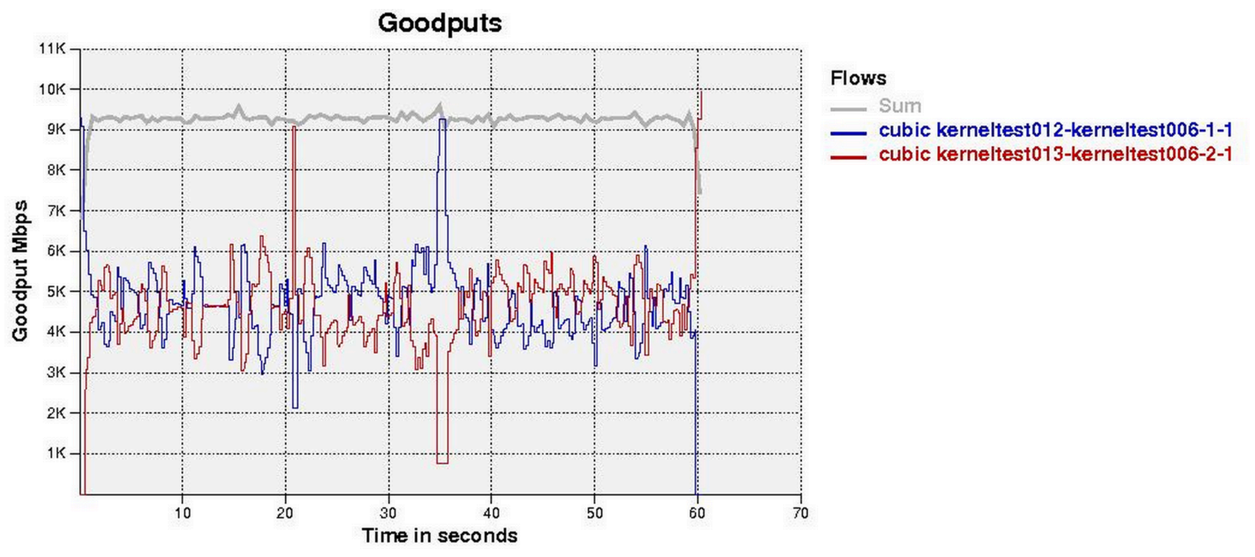


Figure 2a: Two Cubic stream flows, Goodput graph

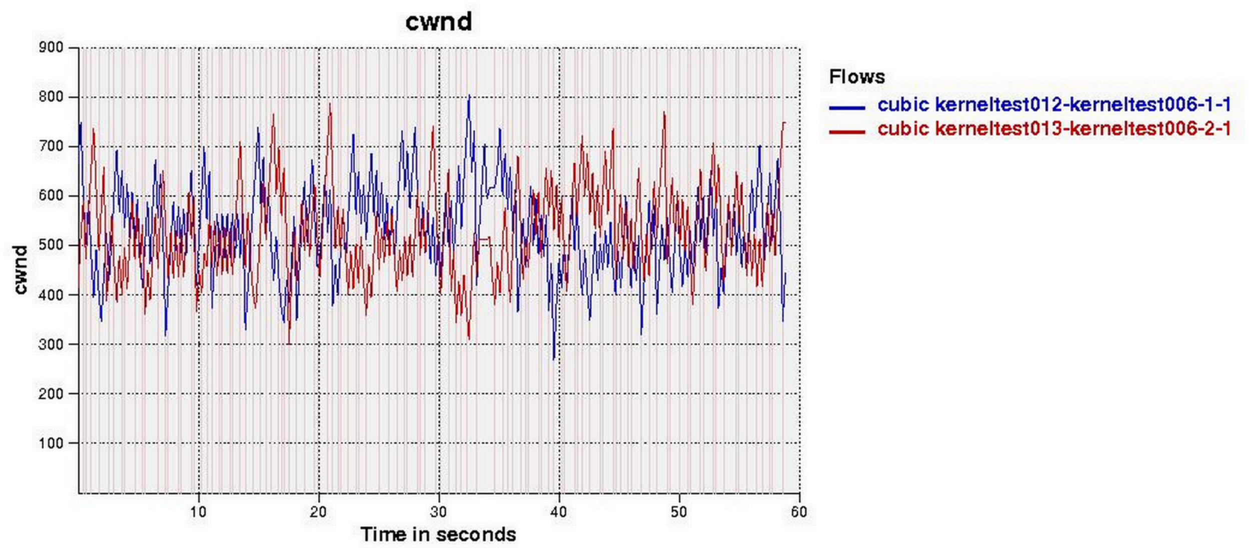


Figure 2b: Two Cubic stream flows, cwnd graph

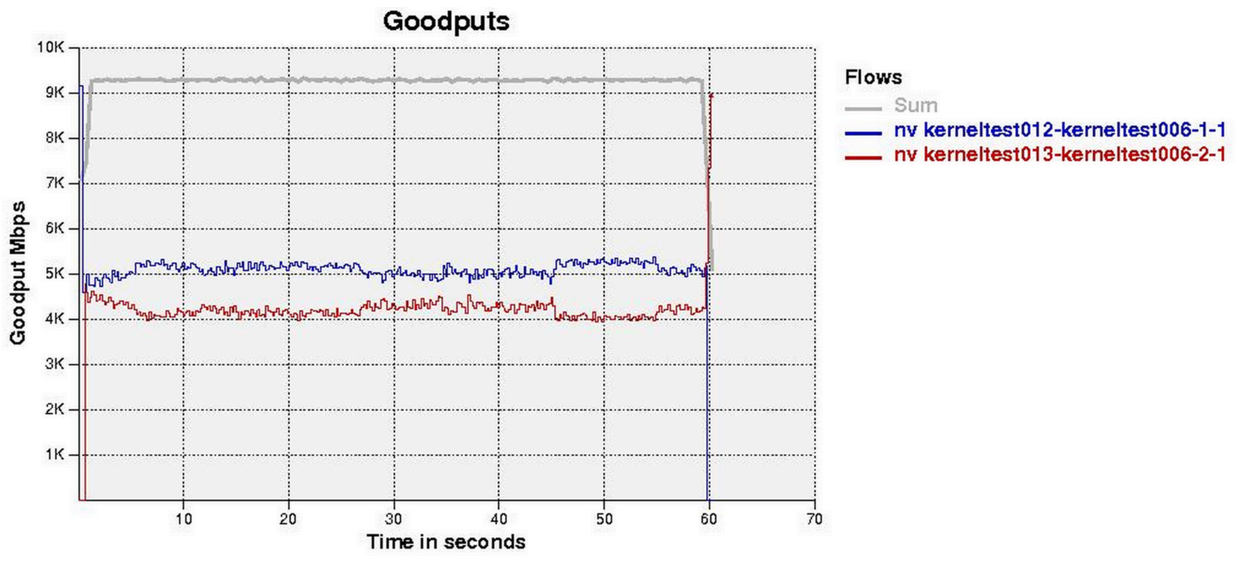


Figure 3a: Two NV stream flows, Goodput graph

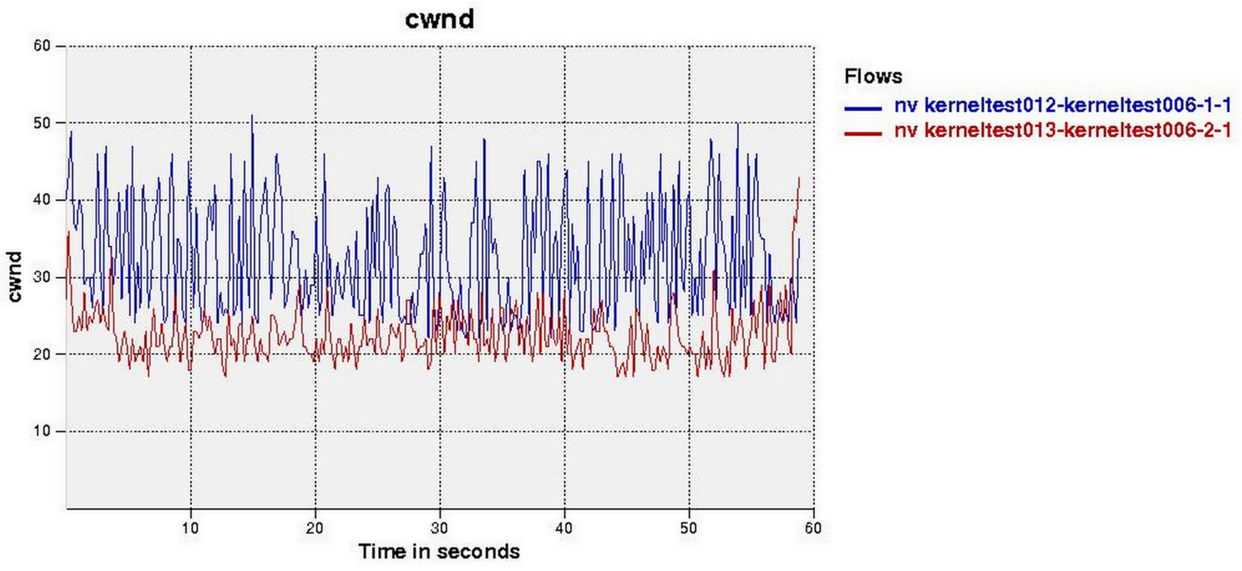


Figure 3b: Two NV stream flows, cwnd graph

The unfairness in NV is due to the different views of minrtt for each flow. The temporary decrease of cwnd was introduced to help the minrtt values converge, but was turned off by default because of its effect on P99 latency. I am exploring various ways to improve this issue.

Figure 3c plots minrtt for both NV flows.

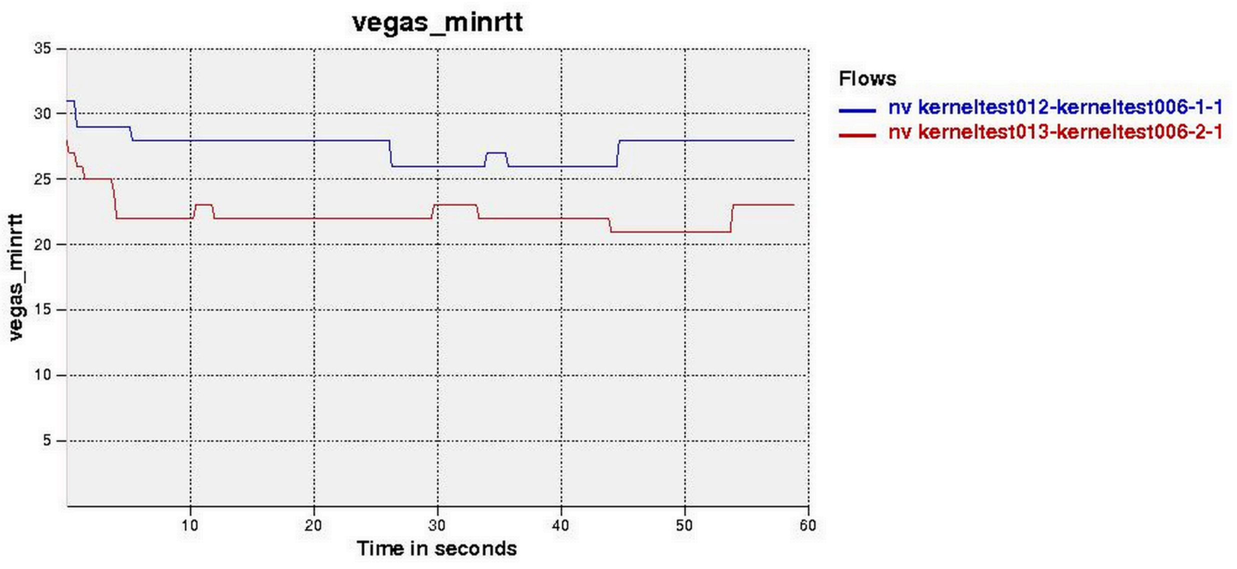


Figure 3b: min-rtt as seen by each NV flow

4.3 2-Clients and 1-Flow per client (partial)

Figures 4 and 5 show how an existing flow adapts when a new flow is started. Figure 4 shows that Cubic adapts well, but once again, it is fair at a larger time scale. It also shows that cwnd is much larger than necessary leading to packet drops (red vertical barss). Figure 5b shows how the first NV flow decreases its cwnd (even though there are no losses) when the new flow starts since now it has less bandwidth available for it.

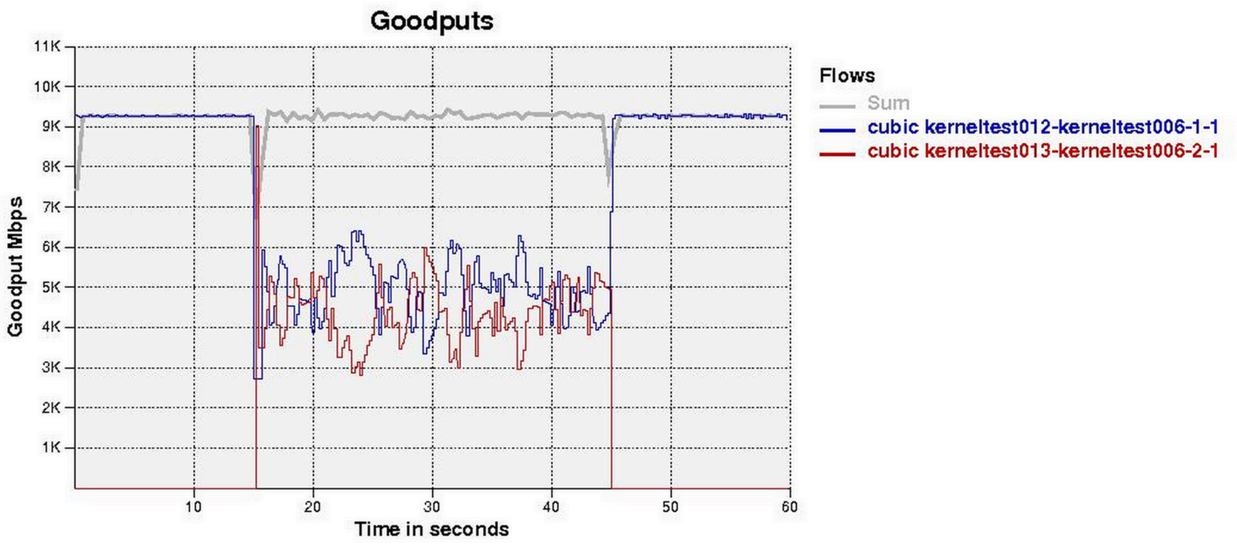


Figure 4a: Cubic adaptation to a new flow, Goodput graph

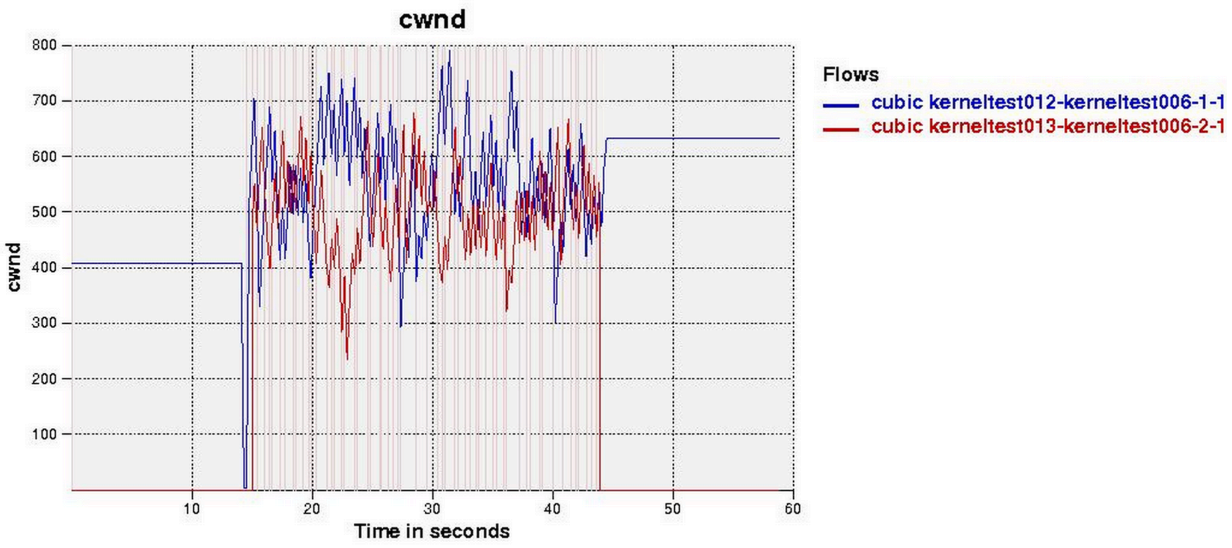


Figure 4b: Cubic adaptation to a new flow, cwnd graph

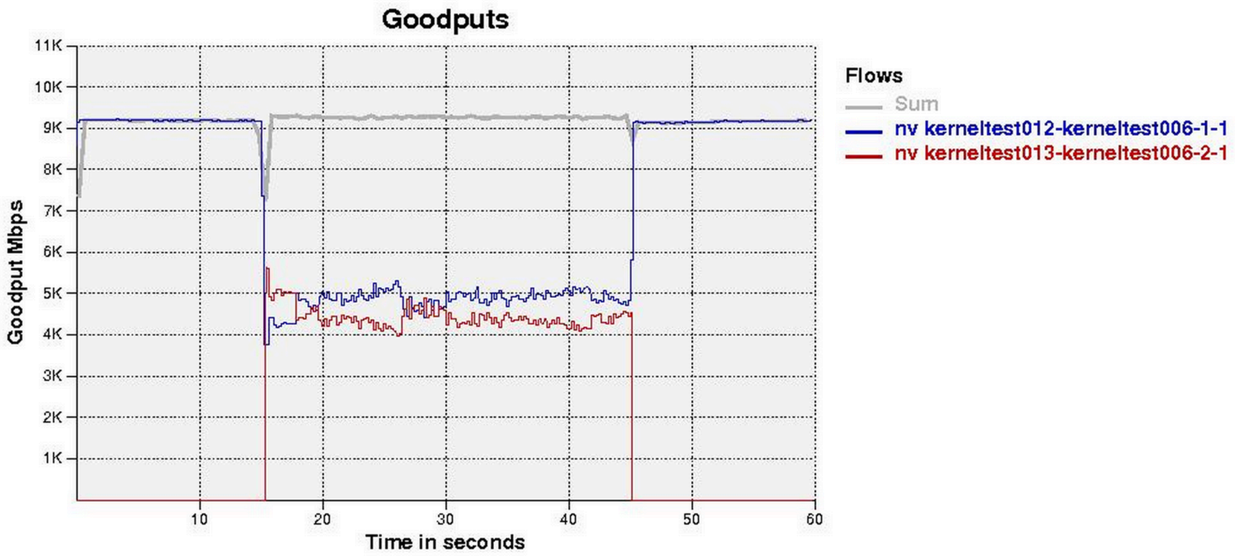


Figure 5a: NV adaptation to a new flow, Goodput graph

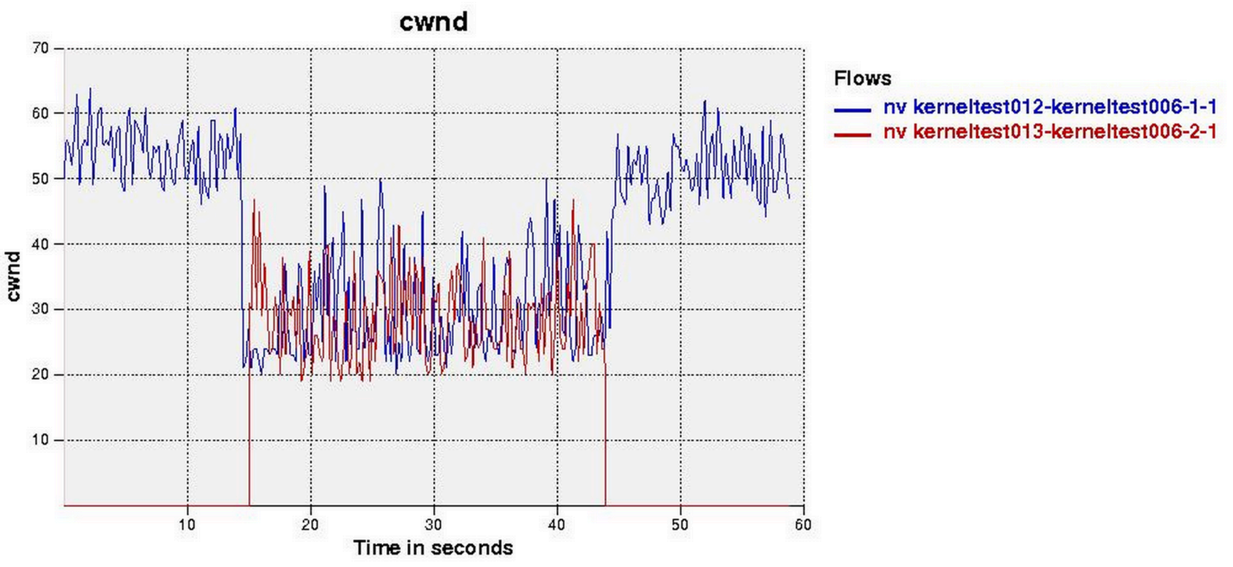


Figure 5b: NV adaptation to a new flow, cwnd graph

4.4 10K vs 1M RPCs

This section explores the effect that large RPCs have on smaller RPCs. In particular, we examine the case of 2 server and 3 clients, with each client having 5 RPC flows to each server. For each server 1 of the RPCs is a 10K RPC, the other 4 are 1MB RPCs.

Table 3 shows that Reno/Cubic/CDG give most of the bandwidth to the 1MB RPCs. The 1MB RPCs average rates of ~ 1.5Gbps, while the 10KB RPCs only average 82Mbps. In contrast, NV achieves 10KB RPCs rates of 480Mbps (almost 6x better) and 1MB RPC rates of 1.4 Gbps. Note that one 10KB RPC, by itself, can only achieve 2Gbps in our scenario since it is only sending 10KB per RTT.

As a result, NV 10KB RPC P99 latencies are 231us vs. 1.6ms for Reno and Cubic and 2ms for CDG. Average latencies are 171us for NV vs. ~1ms for the others.
Even though NV allocates less bandwidth to 1MB RPCs, their P99 latencies are smaller than the others (8.6ms vs. 9.5/9.0/10.2ms). However, NV 1MB average RPC latencies are slightly larger: 6ms vs. 5.5/5.3/5.3ms.

All TCP variants achieve similar link utilization with aggregate goodputs around 19.6 Gbps (we have 2 servers receiving data).

Chart 2 shows the mean and P99 latencies for 10KB and 1MB RPC sizes. NV’s 10KB RPC mean latencies are 83% lower than those of Reno/Cubic/CDG. The P99 latency of NV is similar to its mean latency while those of the other TCP variants are 50-100% larger.

	exp	expName	test	ca	req	cwnd	rtt	rate	retransPkts%	meanLatency	p99Latency	retrans_total
	1025.0	2s3c3fvr	TCP_RR	reno	1M+...	125.7	1313.0	19519.0	0.0	3907.0	6869.0	13817.0
	1025.10	2s3c3fvr	TCP_RR	reno	1M	184.5	1434.0	1585.0		5369.0	9520.0	1130.0
	1025.12	2s3c3fvr	TCP_RR	reno	10K	8.16	1071.0	82.83		984.9	1569.0	41.16
	1026.0	2s3c3fvr	TCP_RR	cubic	1M+...	130.6	1282.0	19555.0	0.0	3876.0	6562.0	11948.0
	1026.10	2s3c3fvr	TCP_RR	cubic	1M	191.7	1367.0	1590.0		5298.0	9041.0	980.5
	1026.12	2s3c3fvr	TCP_RR	cubic	10K	8.33	1111.0	79.16		1034.0	1604.0	30.33
	1027.0	2s3c3fvr	TCP_RR	nv	1M+...	16.7	308.6	19745.0	0.0	4044.0	5806.0	0.0
	1027.10	2s3c3fvr	TCP_RR	nv	1M	19.66	169.0	1406.0		5981.0	8594.0	0.0
	1027.12	2s3c3fvr	TCP_RR	nv	10K	10.83	160.5	478.8		170.6	230.6	0.0
	1028.0	2s3c3fvr	TCP_RR	cdg	1M+...	120.1	1226.0	19674.0	0.0	3831.0	7446.0	9205.0
	1028.10	2s3c3fvr	TCP_RR	cdg	1M	176.0	1307.0	1598.0		5253.0	10177.0	758.6
	1028.12	2s3c3fvr	TCP_RR	cdg	10K	8.0	1063.0	82.83		985.1	1984.0	16.83

Table 3: 10K vs. 1MB RPCs

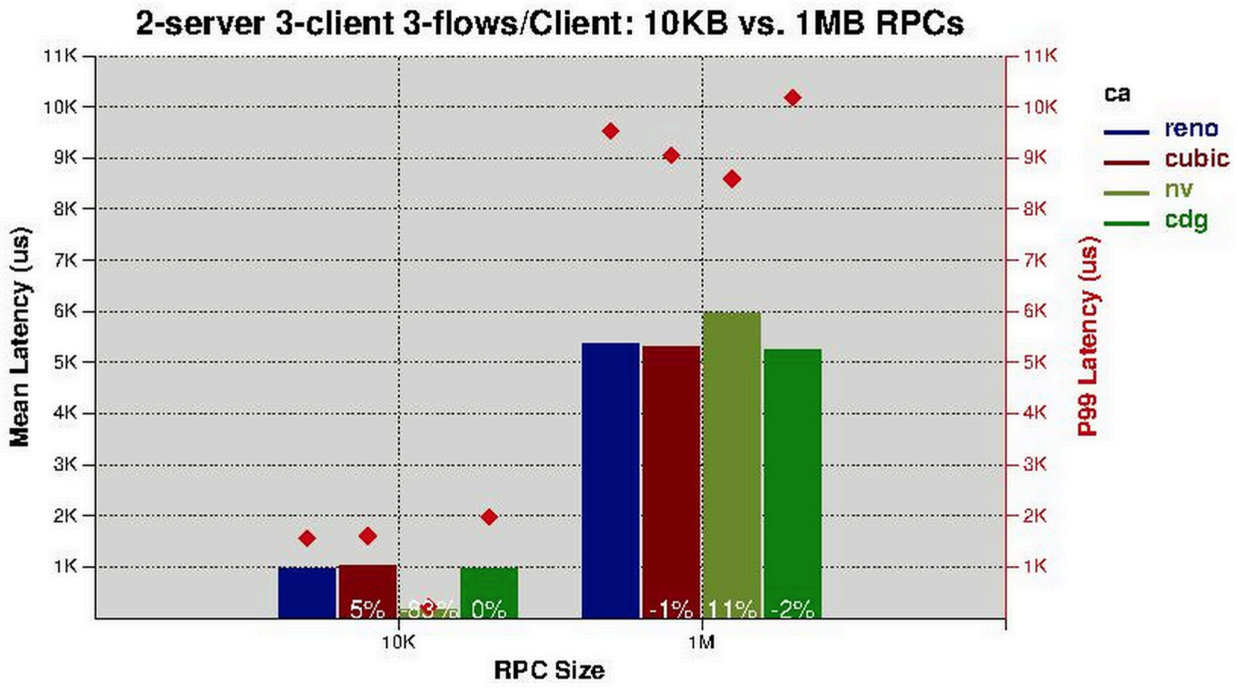
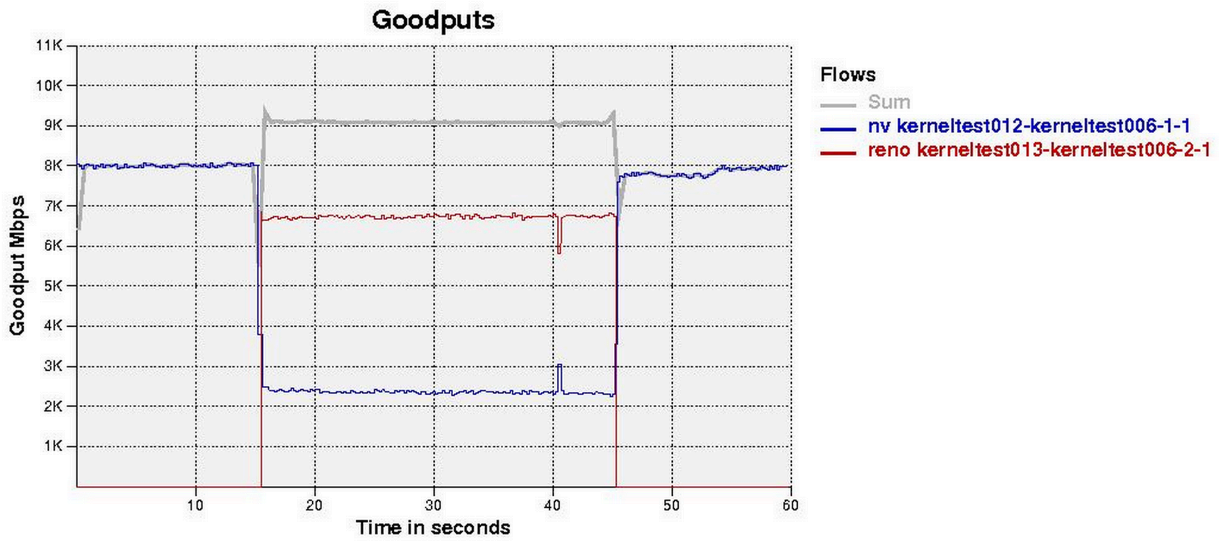


Chart 2: 10KB vs. 1MB RPC Latencies

4.5 Reno/Cubic vs. NV

Figure 6 what happens when a Reno flow is started in the middle of an NV flow (both 1MB RPCs). As mentioned earlier, congestion avoidance and congestion control result in bad unfairness for congestion avoidance unless appropriate mechanisms are used (e.g. isolating the flows). The graph looks similar for Cubic vs. NV.

There are no fairness issues between Reno, Cubic and CDG.



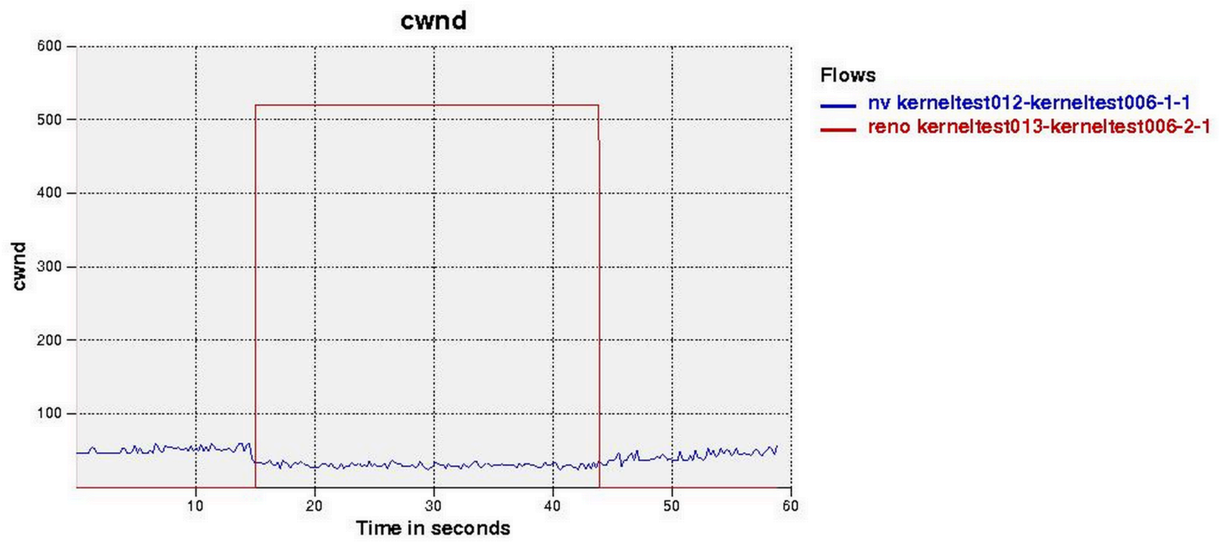
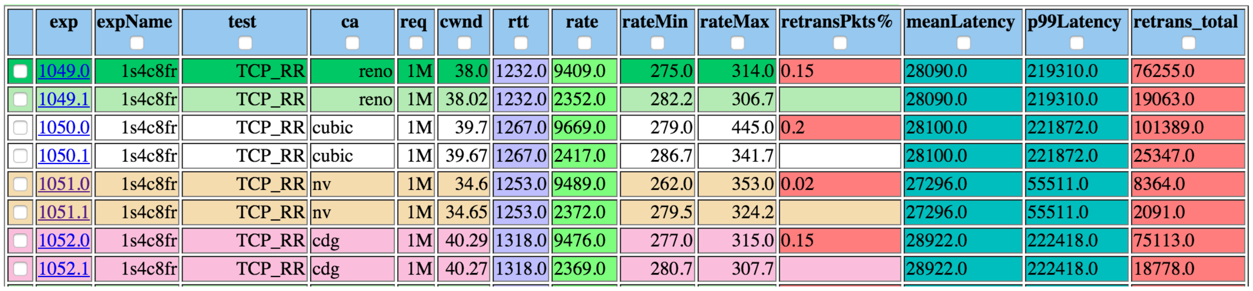


Figure 6: Cubic vs. NV

4.6 4-Clients and 8 flows per client sending to 1 server

Table 4 shows what happens when there are 4 clients and 8 1MB RPC flows per client sending to 1 server. The .0 results show the average cwnd and rtt, the aggregate goodput and the min and max rates as well as the % packets retransmitted. The .1 results show the average goodput per client and the average min and max. All TCP variants have losses, even NV cannot prevent them due to the number of flows (32) and min_rtt inflation. However NV has 7-10x fewer losses and P99 latencies are 4x smaller.



	exp	expName	test	ca	req	cwnd	rtt	rate	rateMin	rateMax	retransPkts%	meanLatency	p99Latency	retrans_total
☐	1049.0	1s4c8fr	TCP_RR	reno	1M	38.0	1232.0	9409.0	275.0	314.0	0.15	28090.0	219310.0	76255.0
☐	1049.1	1s4c8fr	TCP_RR	reno	1M	38.02	1232.0	2352.0	282.2	306.7		28090.0	219310.0	19063.0
☐	1050.0	1s4c8fr	TCP_RR	cubic	1M	39.7	1267.0	9669.0	279.0	445.0	0.2	28100.0	221872.0	101389.0
☐	1050.1	1s4c8fr	TCP_RR	cubic	1M	39.67	1267.0	2417.0	286.7	341.7		28100.0	221872.0	25347.0
☐	1051.0	1s4c8fr	TCP_RR	nv	1M	34.6	1253.0	9489.0	262.0	353.0	0.02	27296.0	55511.0	8364.0
☐	1051.1	1s4c8fr	TCP_RR	nv	1M	34.65	1253.0	2372.0	279.5	324.2		27296.0	55511.0	2091.0
☐	1052.0	1s4c8fr	TCP_RR	cdg	1M	40.29	1318.0	9476.0	277.0	315.0	0.15	28922.0	222418.0	75113.0
☐	1052.1	1s4c8fr	TCP_RR	cdg	1M	40.27	1318.0	2369.0	280.7	307.7		28922.0	222418.0	18778.0

Table 4: Four Clients and 8 1MB RPC flows per client

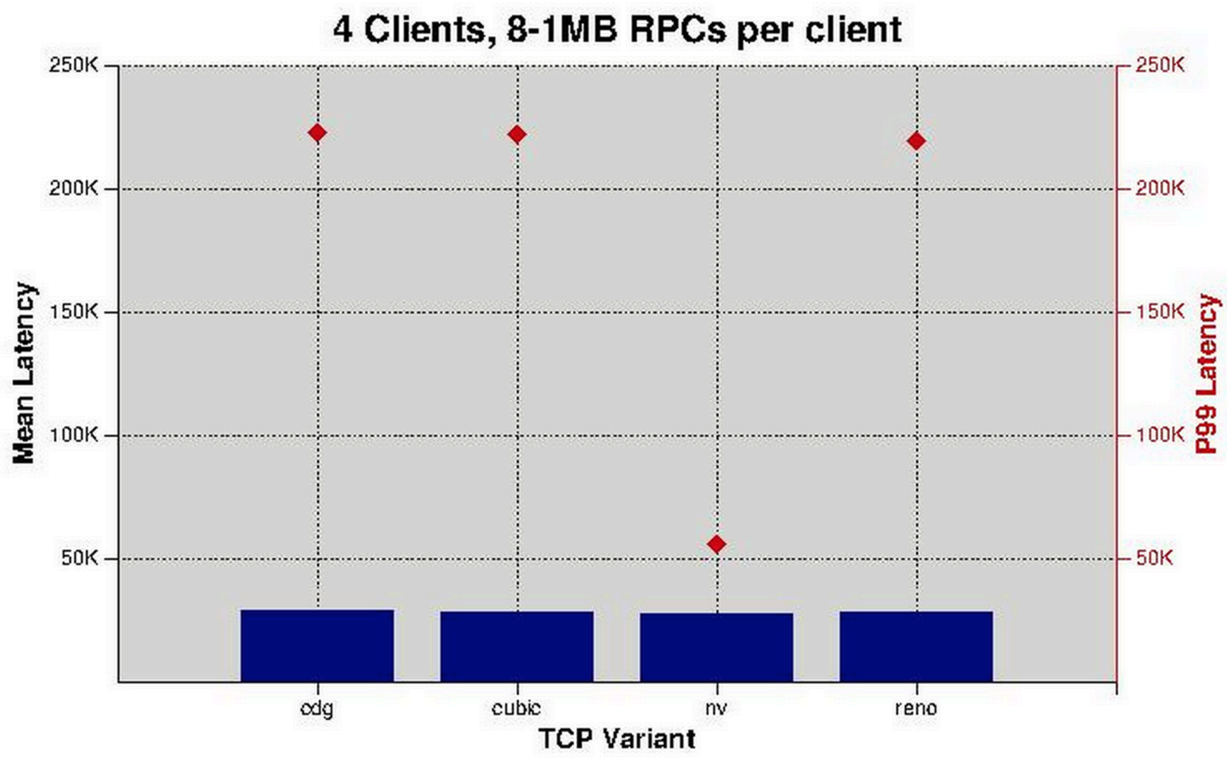
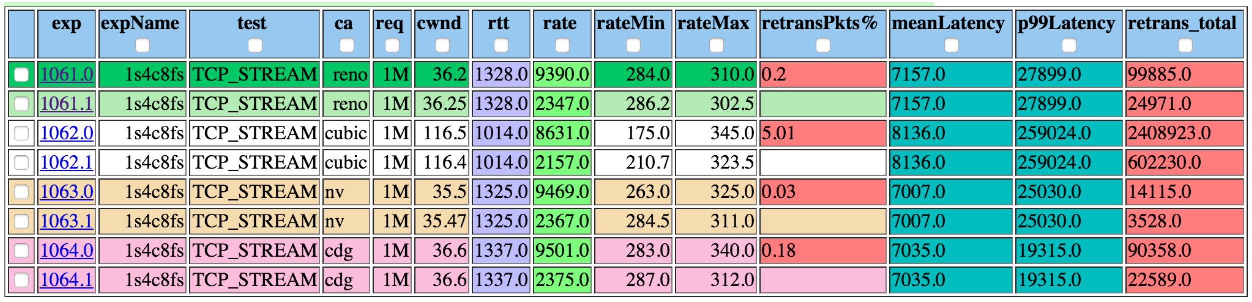


Chart 3: Mean and P99 Latencies for 32 1MB RPCs

Table 5 shows the results for STREAM. Cubic now has much higher losses, 5% of the packets are retransmitted. As a result, its P99 latency is almost 10x higher as compared to Reno/NV/CDG (remember that STREAM latency is the duration of the write call). NV has 7x fewer retransmission than Reno and CDG.

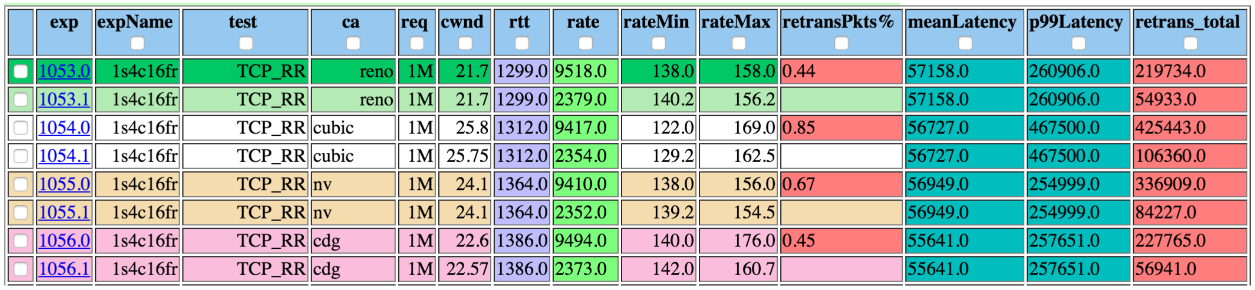


	exp	expName	test	ca	req	cwnd	rtt	rate	rateMin	rateMax	retransPkts%	meanLatency	p99Latency	retrans_total
☐	1061.0	1s4c8fs	TCP_STREAM	reno	1M	36.2	1328.0	9390.0	284.0	310.0	0.2	7157.0	27899.0	99885.0
☐	1061.1	1s4c8fs	TCP_STREAM	reno	1M	36.25	1328.0	2347.0	286.2	302.5		7157.0	27899.0	24971.0
☐	1062.0	1s4c8fs	TCP_STREAM	cubic	1M	116.5	1014.0	8631.0	175.0	345.0	5.01	8136.0	259024.0	2408923.0
☐	1062.1	1s4c8fs	TCP_STREAM	cubic	1M	116.4	1014.0	2157.0	210.7	323.5		8136.0	259024.0	602230.0
☐	1063.0	1s4c8fs	TCP_STREAM	nv	1M	35.5	1325.0	9469.0	263.0	325.0	0.03	7007.0	25030.0	14115.0
☐	1063.1	1s4c8fs	TCP_STREAM	nv	1M	35.47	1325.0	2367.0	284.5	311.0		7007.0	25030.0	3528.0
☐	1064.0	1s4c8fs	TCP_STREAM	cdg	1M	36.6	1337.0	9501.0	283.0	340.0	0.18	7035.0	19315.0	90358.0
☐	1064.1	1s4c8fs	TCP_STREAM	cdg	1M	36.6	1337.0	2375.0	287.0	312.0		7035.0	19315.0	22589.0

Table 5: Four clients and 8 STREAM flows per client

4.7 4-Clients and 16 flows per client sending to 1 server

Table 6 shows the results when there are 16 1MB RPCs per client. Cubic has the worst P99 latency, almost twice as large as for the others. NV has higher losses than Reno and CDG but the P99 latencies are similar. With 64 flows going to one receiver, NV's min_rtt grows quite large and is not able to prevent congestion.



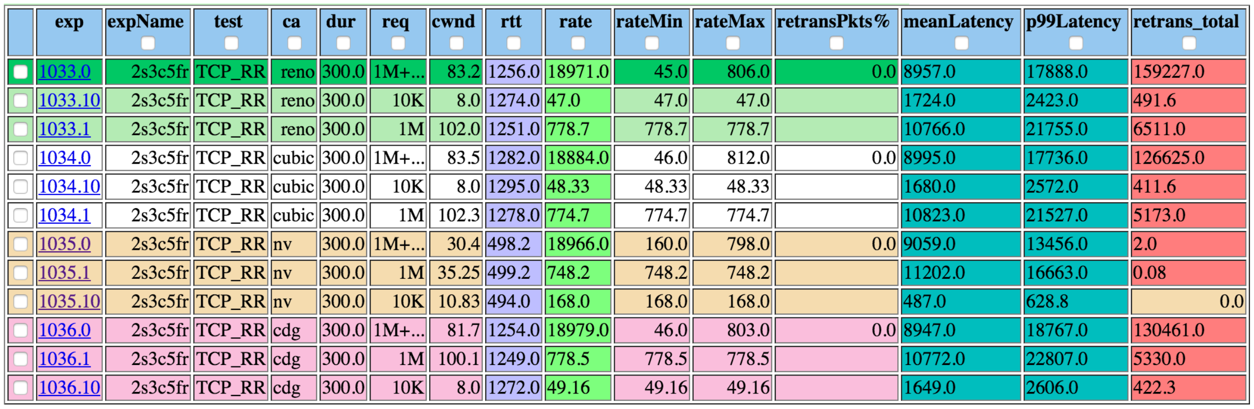
	exp	expName	test	ca	req	cwnd	rtt	rate	rateMin	rateMax	retransPkts%	meanLatency	p99Latency	retrans_total
☐	1053.0	1s4c16fr	TCP_RR	reno	1M	21.7	1299.0	9518.0	138.0	158.0	0.44	57158.0	260906.0	219734.0
☐	1053.1	1s4c16fr	TCP_RR	reno	1M	21.7	1299.0	2379.0	140.2	156.2		57158.0	260906.0	54933.0
☐	1054.0	1s4c16fr	TCP_RR	cubic	1M	25.8	1312.0	9417.0	122.0	169.0	0.85	56727.0	467500.0	425443.0
☐	1054.1	1s4c16fr	TCP_RR	cubic	1M	25.75	1312.0	2354.0	129.2	162.5		56727.0	467500.0	106360.0
☐	1055.0	1s4c16fr	TCP_RR	nv	1M	24.1	1364.0	9410.0	138.0	156.0	0.67	56949.0	254999.0	336909.0
☐	1055.1	1s4c16fr	TCP_RR	nv	1M	24.1	1364.0	2352.0	139.2	154.5		56949.0	254999.0	84227.0
☐	1056.0	1s4c16fr	TCP_RR	cdg	1M	22.6	1386.0	9494.0	140.0	176.0	0.45	55641.0	257651.0	227765.0
☐	1056.1	1s4c16fr	TCP_RR	cdg	1M	22.57	1386.0	2373.0	142.0	160.7		55641.0	257651.0	56941.0

Table 6: Four Clients and 16 1MB RPC flows per client

4.8 5 minute experiments

We also ran longer experiments, 5 minutes long, to verify whether NV can maintain control of cwnd and minRTT when it runs for a longer time or whether it continues to grow. The experiment consists of 3 clients sending 1-10KB and 4-1MB RPCs to 2 servers (30 flows).

Table 7 shows the results for Reno/Cubic/NV/CDG. NV P99 latency for 10KB RPCs is about 4x lower than the others, mean latency is about 3x lower. There were only 2 retransmitted packets for NV versus >120,000 for the others (although % retrans were less than 0.01%)



	exp	expName	test	ca	dur	req	cwnd	rtt	rate	rateMin	rateMax	retransPkts%	meanLatency	p99Latency	retrans_total
☐	1033.0	2s3c5fr	TCP_RR	reno	300.0	1M+...	83.2	1256.0	18971.0	45.0	806.0	0.0	8957.0	17888.0	159227.0
☐	1033.10	2s3c5fr	TCP_RR	reno	300.0	10K	8.0	1274.0	47.0	47.0	47.0		1724.0	2423.0	491.6
☐	1033.1	2s3c5fr	TCP_RR	reno	300.0	1M	102.0	1251.0	778.7	778.7	778.7		10766.0	21755.0	6511.0
☐	1034.0	2s3c5fr	TCP_RR	cubic	300.0	1M+...	83.5	1282.0	18884.0	46.0	812.0	0.0	8995.0	17736.0	126625.0
☐	1034.10	2s3c5fr	TCP_RR	cubic	300.0	10K	8.0	1295.0	48.33	48.33	48.33		1680.0	2572.0	411.6
☐	1034.1	2s3c5fr	TCP_RR	cubic	300.0	1M	102.3	1278.0	774.7	774.7	774.7		10823.0	21527.0	5173.0
☐	1035.0	2s3c5fr	TCP_RR	nv	300.0	1M+...	30.4	498.2	18966.0	160.0	798.0	0.0	9059.0	13456.0	2.0
☐	1035.1	2s3c5fr	TCP_RR	nv	300.0	1M	35.25	499.2	748.2	748.2	748.2		11202.0	16663.0	0.08
☐	1035.10	2s3c5fr	TCP_RR	nv	300.0	10K	10.83	494.0	168.0	168.0	168.0		487.0	628.8	0.0
☐	1036.0	2s3c5fr	TCP_RR	cdg	300.0	1M+...	81.7	1254.0	18979.0	46.0	803.0	0.0	8947.0	18767.0	130461.0
☐	1036.1	2s3c5fr	TCP_RR	cdg	300.0	1M	100.1	1249.0	778.5	778.5	778.5		10772.0	22807.0	5330.0
☐	1036.10	2s3c5fr	TCP_RR	cdg	300.0	10K	8.0	1272.0	49.16	49.16	49.16		1649.0	2606.0	422.3

Table 7: 2-server 3-clients and 1-10KB and 4-1MB RPCs per Client

Figure 7 shows the min_rtt as calculated by each of the 30 flows. It grows for the 1st 50 seconds, but then stabilizes to 300-450us. Quite larger than the true HW min_rtt of ~30us, but it still allows NV to prevent packet losses (there is only one loss at around 60 seconds).

Figure 8 shows that NV still manages to control cwnd growth without losses, although the values are much higher than ideal.

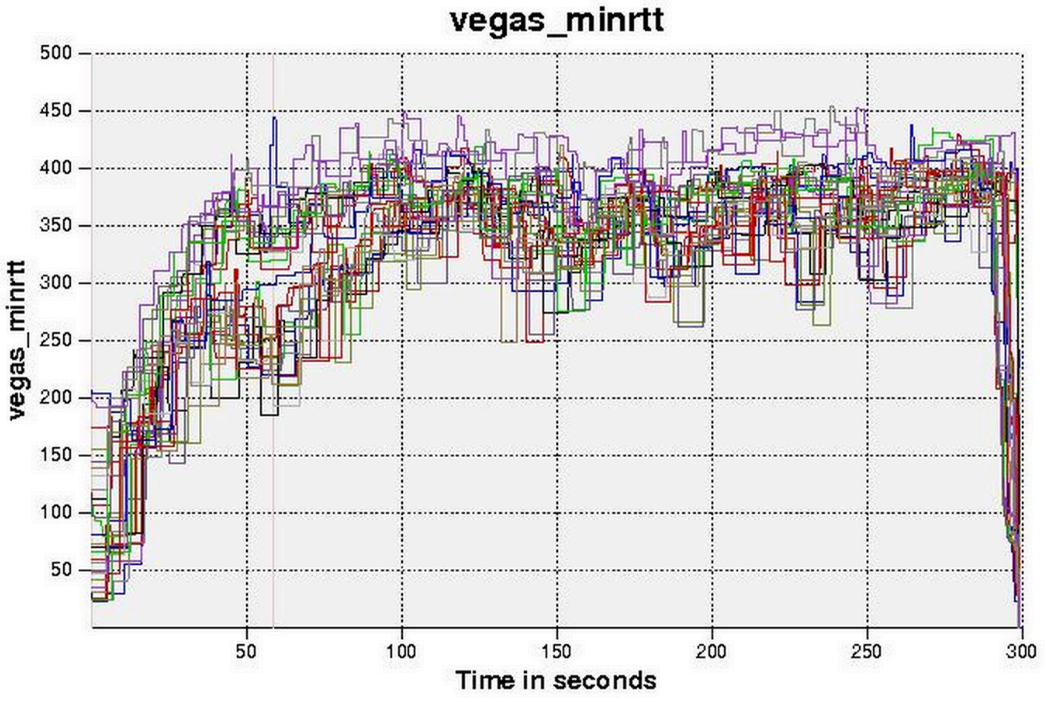


Figure 7: min_rtt for each of the 60 flows

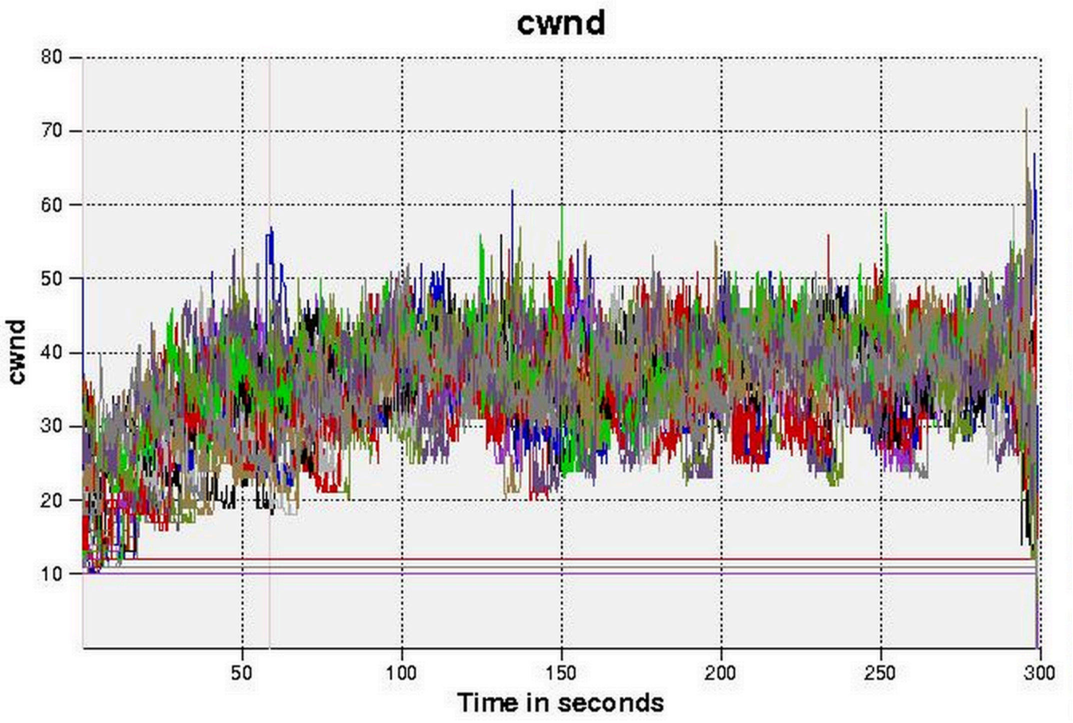


Figure 8: cwnd for NV

5. Future Work

There are a couple of things planned for the future

- **Reverse Congestion**: Congestion seen by the ACKs as opposed to congestion seen by the data packets. Since the ACKs are small compared to data packets, i.e. they are not really congesting the network, congestion seen by the ACKs should be ignored. We currently only see the overall RTT and cannot differentiate between forward or reverse congestion. I have an idea for handling reverse congestion that will be implementing in the future.
- **Better Fairness mechanism**: NV would probably do a lot better if the average link utilization is <100% since this may allow it to get good min_rtt measurements. I still plan to further explore the temporary cwnd reduction mechanism.
- **Complex Experiments**: The experiments I have done since the overhaul of the code has been with small RTTs within a rack. I will be running more complex scenarios and will update the document when that is done.
- **Comparisons:** I plan compare with even more TCP variants, including DC-TCP.