Analysis and Evaluation of TCP Congestion Algorithms

Lawrence Brakmo

Facebook
Menlo Park, USA
brakmo@fb.com

Abstract

The development of new congestion algorithms for TCP has continued for almost 30 years. Starting with Van Jacobson's seminal work on congestion avoidance and control[8] in 1988 to the development of BBR[13][11] in 2016, we have seen many interesting approaches to the basic problem of how to best use the available bandwidth. The fact that so much work has occurred in this area attests to the inherent difficulty of the problem.

In this paper, we analyze and evaluate some of the most relevant TCP congestion algorithms available in Linux. We use Cubic[14] as the baseline since it is the current default congestion algorithm. In addition to Cubic, we have chosen the following congestion algorithms to evaluate: Reno, DCTCP[12], NV[5], and BBR.

Keywords

TCP, Linux, congestion algorithms, Cubic, BBR, DCTCP, NV

Introduction

The basic problem that a congestion algorithm attempts to solve is how to fully, and fairly, utilize the available bandwidth in a connection's path. Early works in this area analyzed the simplest cases, where there were only one or two flows, and very little noise in the measurements. Under these simplifications they were able to develop algorithms that behaved well in theory.

The problem is that reality is seldom sympathetic to theory. As a result, many of these algorithms were never used in actual systems. The problems that a congestion algorithm tries to solve are very hard. The fact that in actual practice there will be many unrelated flows makes the problem even harder. At its most basic, a congestion algorithm is a mechanism so that many unrelated flows can use all of the available bandwidth without overshooting it so much that queues fill up, increasing latency and creating packet drops.

For example, consider when a new flow starts and it has no knowledge of the available bandwidth. It has to probe the network in order to find the available bandwidth. TCP uses slow-start, doubling the size of the congestion window (cwnd) every RTT, for this purpose. That is, TCP will start with a "slow" exponential growth in the cwnd (and rate as long as there is enough bandwidth). Slow-start typically ends with large packet losses. Congestion avoidance mechanisms such as those in NV, DCTCP, BBR or hystart in Cubic can reduce or prevent these losses.

Now, suppose the flow could "magically" know the available bandwidth. What should it do? Should it just start immediately at that rate? What if there were 2 flows starting at the same time and both try to use the available bandwidth at the same time? What if there were 10 flows? The problem is hard even knowing the available bandwidth.

In addition, because packet drops are a fact of life in most real networks, the congestion algorithm must also provide mechanisms to detect these loses as quickly as possible so that it can recover from them.

When considering congestion algorithms, it is important to differentiate between congestion control and congestion avoidance algorithms. But before discussing the difference between the two we need to define congestion. Congestion is the condition that occurs when a network is carrying so much data that it leads to large standing queues. These queues lead to increase latency and, as the condition worsens, to packet losses. Congestion is not only packet losses, but also large queues.

Congestion control algorithms do not try to avoid congestion, they try to control it so that it doesn't cause too much damage. For example, consider the original congestion control in TCP Reno. This original congestion algorithm in used an additive increase and multiplicative decrease (AIMD) of the congestion window (cwnd). The congestion window, representing the number of packets in transit, would increase linearly until packet losses were detected, at which point the congestion window would be decreased by half (resulting in the well now saw tooth graphs).

Reno has no mechanism to detect when all of the available bandwidth was being utilized. As a result, it would increase its cwnd until the bottleneck link was fully utilized, then it would keep increasing it, filling the bottleneck buffers until they overflow and packets were dropped. Reno would detect this packet losses and interpret them as a sign of congestion. It would then decrease its cwnd by half and start the process again. Reno was never avoiding congestion, instead it was periodically creating it and controlling it. It was avoiding congestion collapse, but it was not avoiding congestion.

In contrast, delay based congestion avoidance algorithms like TCP-Vegas[2] would stop growing the congestion window once they detected that all of the available bandwidth was in use.

Of the congestion algorithms we are evaluating in this paper, Reno and Cubic are congestion control algorithms while DCTCP, NV and BBR are congestion avoidance algorithms.

Experimental Setup

The experiments can be broken into 2 orthogonal dimensions. One of these dimensions are experimental scenarios which describe the topology of the network. This includes link bandwidths, RTTs, buffer sizes, etc. The other dimension are the actual tests which encompass the number of active hosts, the number of flows, flow characteristics such as stream vs. RPC as well as RPC sizes.

In particular, the scenarios and tests were used are:

Scenarios

- LAN with 20us RTT, 10 Gbps - servers in same rack (D.C. scenario)
- WAN with 10ms RTT, 10Gbps – using netem []to introduce delay on the receiver (its sending queue). Using a 20,000 packet buffer for netem. The bottleneck is still a real switch with shallow buffers (1-2 MB). Good scenario to visualize the algorithm's dynamics when looking at 2-3 flows.
- WAN 40ms RTT, 10 and 100Mbps. Reflects connection to user using modern networks. This scenario uses an intermediate host that uses the tbf qdisc to reduce the rate to 10 or 100 Mbps.

Figure 0A shows the topology for the first 2 scenarios where the senders and receivers are connected through a switch. Delay is achieved through Netem in the Receiver so it only affects the ACKs being sent back. Figure 0B shows the topology for the 3rd scenario and links between hosts also go through a rack switch not shown in the figure.

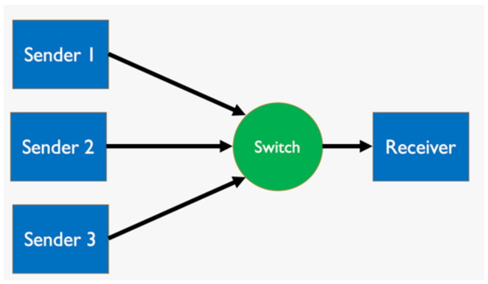

Figure 0A: Topology for 10Gbps tests

Tests:

Fairness & Stability - consists of 2 or 3 stream flow tests (each from a different server) to one receiver. It helps to visualize the dynamics of the congestion control (or avoidance), as well as to examine how each TCP variant competes against itself and against Cubic.

- 2-flow tests: the 1st flow lasts 60 seconds, and the 2nd flow lasts 20 seconds and starts 22 seconds after the 1st one.
- 3-flow tests: the 1st flow lasts 60 seconds, the 2nd flow lasts 40 seconds and starts 12 seconds after the 1st one, the 3rd flow lasts 20 seconds and starts 26 seconds after the 1st one.

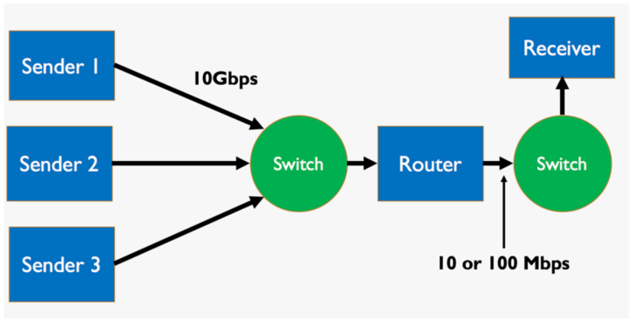

Figure 0B: 10 and 100Mbps toloplogy

Size Fairness – consists of a combination of streaming, 1MB and 10KB RPCs. Are the smaller RPC flows penalized? If so, by how much? This is an important question because smaller RPCs can be more important than bulk transfers (of course QOS can help with this).

Netesto[4] was used to run the experiments, collect the data and produce the graphs and tables. mq and fq_codel with pacing and ECN enabled was used as the queuing discipline on all hosts. All hosts were running Linux kernel version 4.14.0-rc5. The TCP maximum buffer sizes were set to 32MB so as not to affect the behavior of the congestion algorithms. NetEm[15] was used at the recveiver to add network latency.

Interpretation of Tests

The Netesto outputs from all tests include (although these are not necessarily shown here):

- *Goodput* (payload throughput) graphs - These include output for each flow as well as the aggregate goodput.
- *Cwnd* graphs - these include the cwnd for each flow. Red vertical lines indicate the time when one or more packets are retransmitted. they are a visual indicator of when congestion is occurring.
- *RTT* graphs - as seen by each server sending data
- *Graph of cumulative losses per flow*
- *Table of results* - with rows for each server and the aggregate with columns for:
    - test parameters

- average cwnd
- average RTT
- average ping RTT
- rates (avg, min, max)
- latencies (min, avg, max, 50%, 90%, 99% and 99.9%)
- packet retransmissions

2-Flow and 3-Flow Tests

The 2-flow tests are used for:

- examining how quickly existing flow adapt to new flows
- examining how quickly flows adapt to released bandwdith from terminating flows
- fairness between flows with the same congestion algorithm
- fairness between flows using different congestion algorithms
- levels of congestion
- instability - conditions under which TCP variant's performance changes abruptly

Streaming, 1MB and 10KB RPC Tests

These tests are used for:

- Studying the behavior under increasing loads
- Performance (throughput and latency) of 1MB and 10KB flows. How fair is the available bandwidth divided between them?
- instability - conditions when the retransmissions or latency change abruptly

The Congestion Algorithms

As mentioned earlier, we will consider the following algorithms:

- Reno – Although it shares the name with the original congestion algorithm, it has been enhanced through all the enhancements to the basic loss detection mechanisms in Linux. But at its core is still the same AIMD (additive increase, multiplicative decrease) mechanism. When not in slow-start, it increases its cwnd by 1 packet/RTT. When losses are detected (its only mechanism for detecting congestion) it decreases cwnd by 50%.
- Cubic – The default TCP congestion algorithm in Linux. It grows the cwnd logarithmically for cwnd values near the cwnd where losses last occurred, and exponentially (at increasing rates) at other times (it also has a Reno friendly mode that uses additive increase). When losses are detected it decreases cwnd by 30%.
- DCTCP – Data Center TCP uses ECN markings to detect congestion. Unlike the normal TCP response of decreasing cwnd by 50% when an ECN congestion signal is received, DCTCP decreases it based on the proportion of bytes marked in the previous RTT. That is, the response is smaller if 20% of the bytes were marked than if 100% of the bytes are marked. To prevent unfairness against non-ECN traffic (due to Switch behavior), two queues are used. One for ECN and one for non-ECN traffic. As it name implies, can only be used within a DC (small RTTs).
- NV – is a follow-up to Vegas to deal with modern network conditions (larger bandwidths, TSO, LRO, interrupt coalescence, etc.). It uses RTT and achieved throughput to detect queue buildup and its response. It cannot compete against congestion control algorithms (Reno, Cubic, etc) using the same queue. If there will be non-NV traffic, then 2 queues must be used at the switch; one for NV and one for the other flows. It has been developed for the DC and is tuned for that environment. It is also using TCP-BPF to set the baseRTT to 80us.
- BBR – A new congestion avoidance algorithm that also uses RTT and throughput. However, it does not necessarily interpret losses as a sign of congestion. This can create unfairness when competing with other algorithms.
- TCP-BPF – Using congestion algorithms in conjunction with TCP-BPF in order to set cwnd clamp and TCP connection buffer sizes based on the distance between the hosts. We tested various congestion algorithms but we only show Cubic since the results were similar.

LAN Scenario

Consists of all servers within a rack, 10Gbps links and ~20us RTTs. Each experiment ran more than 10 times and graphs were chosen to represent the most common behavior.

2-Flow LAN, all flows same TCP variant

Figure 1A shows the Goodput of Cubic for 2 flows (2nd flow starts in the middle of the 1st one). The 2nd flow stops its slow-start due to losses early leading to unfairness. However, their goodputs would have converged given more time (and assuming no other flows). It could be considered fair at very large timescales. Figure 1B shows the congestion windows. The vertical red bars indicate when packets were retransmitted by either flow.

To achieve full link utilization, we only need a cwnd of 80. So, when cubic starts by itself, the host queue has more than 400 packets. When congestion occurs due to the start of the other flow, the cwnd increases to 2500 packets.

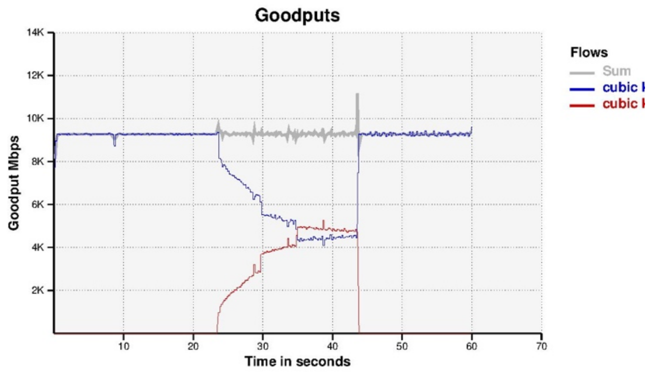

Figure 1A: 2 Goodput of 2 Cubic flows

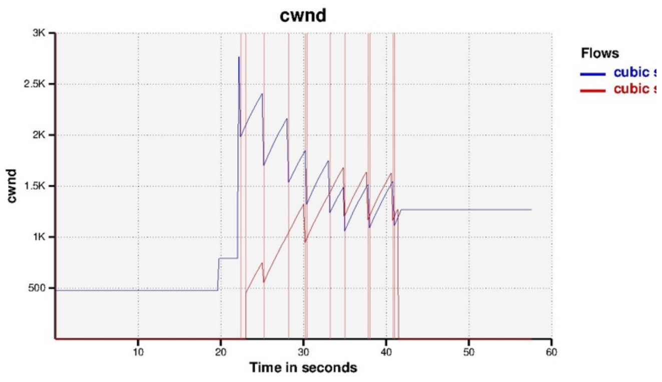

Figure 1B: Cwnd of 2 Cubic flows

Reno behaves similarly so it is not shown.

Figure 2A shows the behavior of DCTCP, which uses ECN markings at the router to achieve congestion avoidance. It is very fair and both flows quickly converge to nearly the same goodput. More interestingly, Figure 2B shows the congestion windows. When there is only one flow, its congestion window is much larger than necessary since there is no congestion at the switch to trigger ECN markings. When the 2nd flow starts, there is congestion and their congestion windows decrease to an optimal size. That is, large enough to fully utilize the bandwidth but not much more to prevent extra latency.

For some unknown reason, it is encountering retransmissions at the start, when there should be no congestion (there is only 1 flow). In the 1st 200ms it retransmits 7000 packets. However, this does not occur in every test. Currently investigating the cause.

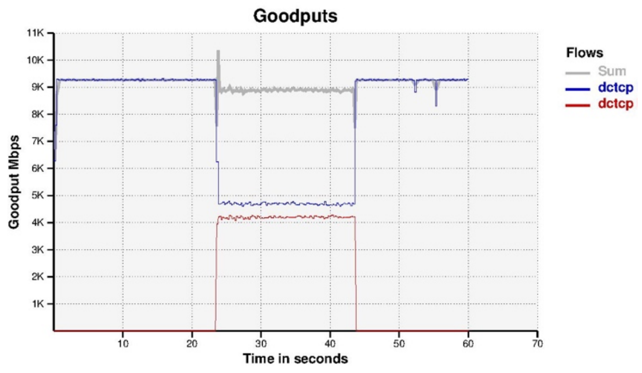

Figure 2A: Goodput for 2 DCTCP flows

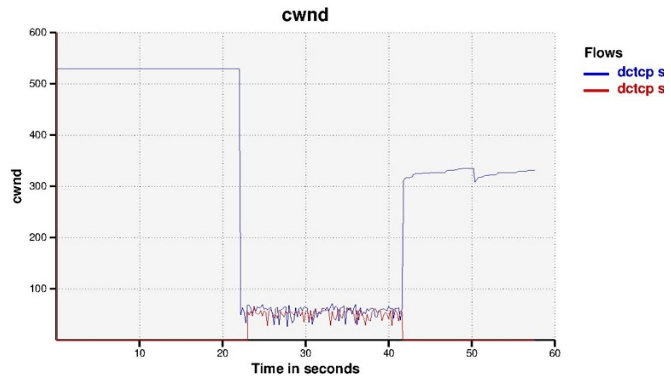

Figure 2B: Cwnd for 2 DCTCP flows

Figure 3A and 3B show similar graphs for BBR. It is quite fair at larger time scale, with small unfairness at the smaller time scales. The congestion windows are not as optimally sizes as with DCTCP.

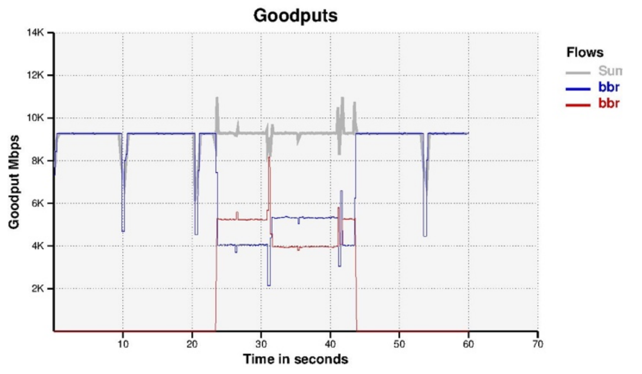

Figure 3A: Goodput for 2 BBR flows

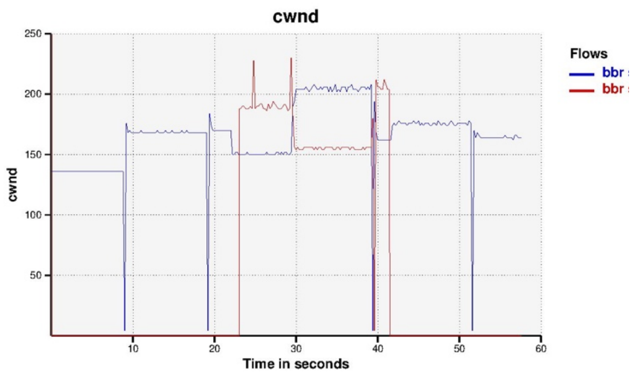

Figure 3B: Cwnd for 2 BBR flows

Next, Figures 4A and B are for NV. Its fairness is similar to DCTCP. With 1 flow, it is more efficient in its use of the cwnd, but with 2 flows DCTCP is more efficient.

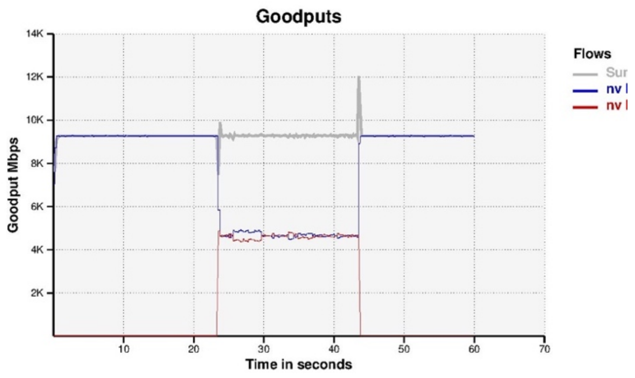

Figure 4A: Goodput for 2 NV flows

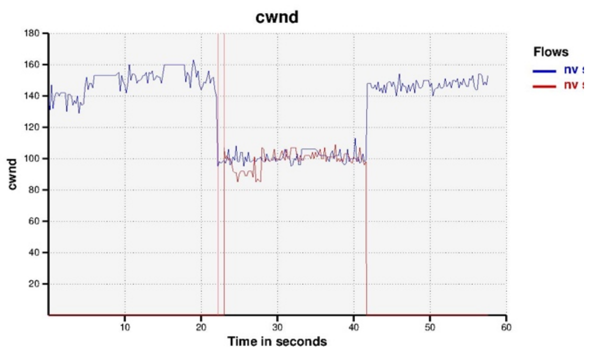

Figure 4B: Cwnds for 2 NV flows

Finally, Figures 5A and 5B show the behavior when Cubic is used with TCP-BPF to set the cwnd clamp to 100 and TCP send and receive buffer sizes to 150KB. The result is perfect fairness and low latency since the cwnds are now 100 per flow instead of around 1500 per flow.

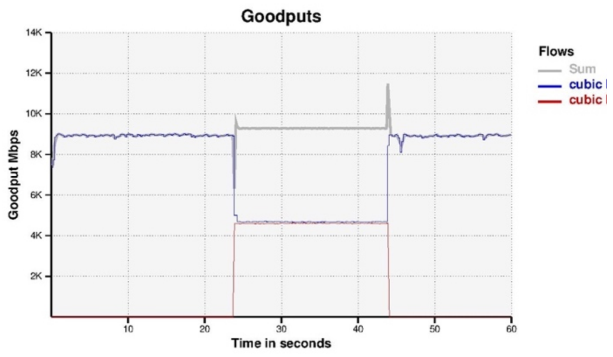

Figure 2A: Goodput for 2 TCP-BPF Cubic flows

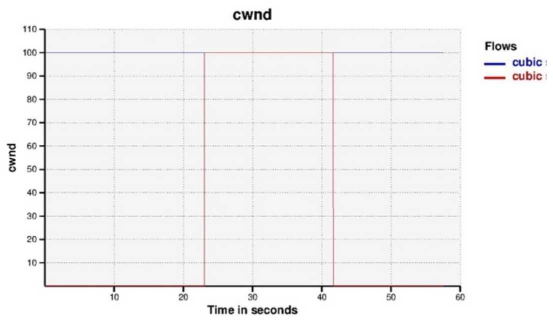

Figure 5B: Cwnds for 2 TCP-BPF Cubic flows

2-Flow LAN, 1st flow Cubic

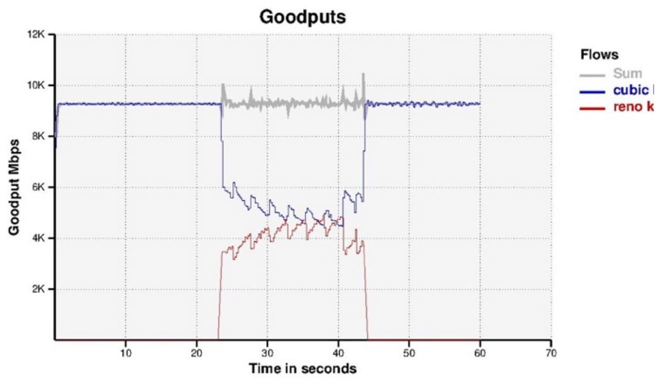

Figure 6A: Goodput of Cubic vs. Reno

Figures 6A to 6C show the behavior when one Cubic flow competes with a non-Cubic flow. Figure 6A shows Cubic and Reno together. On a LAN, Reno and Cubic compete fairly at a coarse timescale. However, it may seem surprising how long it takes for then to converge when the hardware RTTs are only 20us. The reason is that their cwnds are about

1500 so the measured RTTs are 3ms (i.e. it takes 3ms to increase cwnd by one).

Figure 6B shows the goodput of Cubic vs. DCTCP. It achieves as close to perfect fairness. The reason for this is that we have two queues on every port at the switch. One for ECN enabled traffic (DCTCP) and the other for NON-ECN traffic (Cubic). When there is contention, the bandwidth is divided evenly between the queues. However, the cwnds for Cubic are around 2500. This is the same behavior with NV since it also has its own queue.

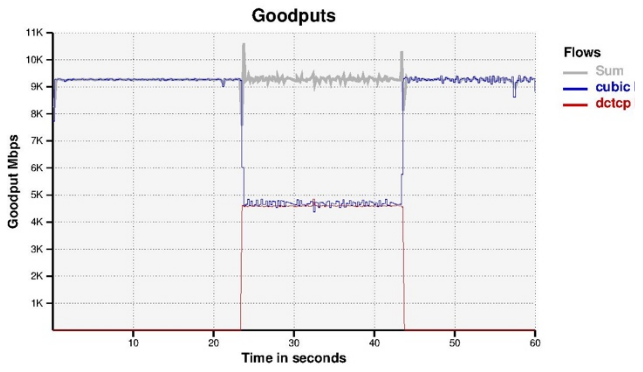

Figure 3B: Goodput of Cubic vs. DCTCP

However, as we will see later, when there are 3 flows with at least one Cubic and one DCTCP, then the bandwidth is divided evenly between ECN and non-ECN flows. Thus, when more flows are in one category than the other, then those flows get less bandwidth individually than those in the other category. For example, when there are 2 DCTCP flows and 1 Cubic flow, the DCTCP flows get 25% of the bandwidth each, while the Cubic flow get 50% of the bandwidth.

Figure 6C shows Cubic vs. BBR. For the 1st half Cubic is using most of the bandwidth. Then 10 seconds after the BBR flow starts and does it RTT and rate probing, it grabs most of the bandwidth. Figure 6D shows the behavior of the congestion windows.

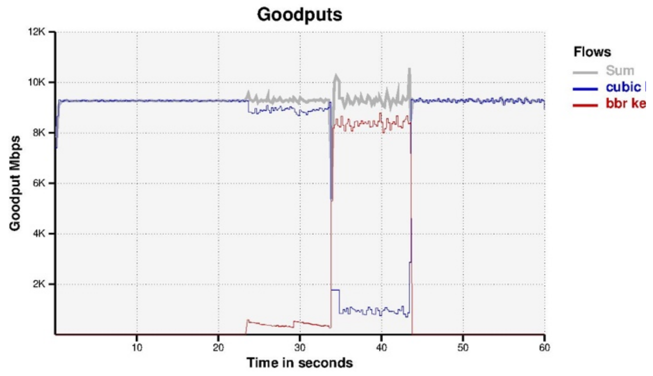

Figure 6C: Goodput of Cubic vs. BBR

When using TCP-BPF applied to all flows, then there is complete fairness and all the cwnds are around 100. And graphs look similar to 5A and 5B.

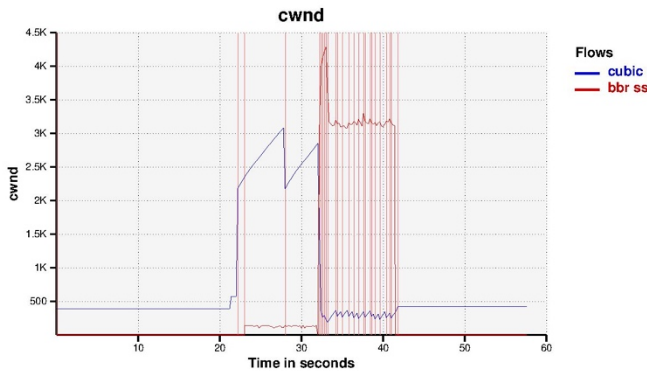

Figure 6D: Cwnds of Cubic and BBR

3-Flow LAN

Results are similar to 2-Flow when all flows are the same, with all algorithms performing much better when TCP-BPF is used (perfect fairness, low cwnds and low buffer usage).

3-Flow LAN, 1st Flow Cubic

When the flows are using 2 different congestion algorithms then there are 3 cases. The 1st case is when the flows are using the same switch queues and not using TCP-BPF. There is unfairness at 5 second time scales. Figure 7A shows Cubic vs BBR. 2nd case is when one of the congestion algorithms has its own queue, like DCTCP or NV. There is perfect sharing between the 2 queues, which means all of the flows in each queue share 50% of the bandwidth.

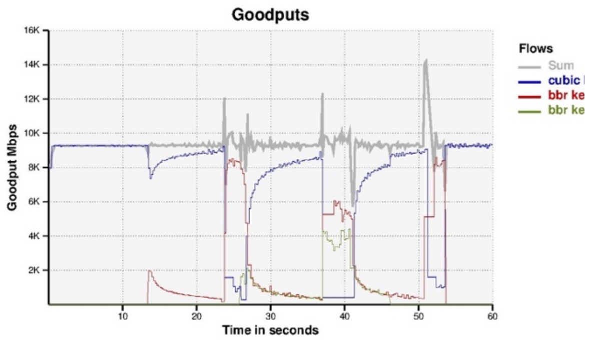

Figure 7A: Cubic vs. 2-BBR flows

Figure 7B shows Cubic vs. DCTCP. Finally, the 3rd case is when TCP-BPF is used. All flows share the bandwidth evenly and their cwnds are small. Figure 7C shows Cubic vs. BBR with TCP-BPF.

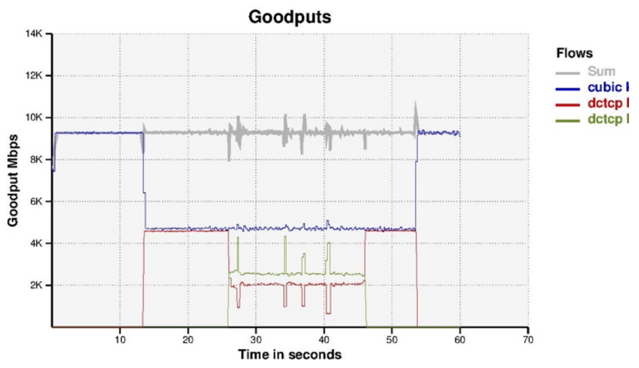

Figure 7B: Cubic vs. 2-DCTCP flows

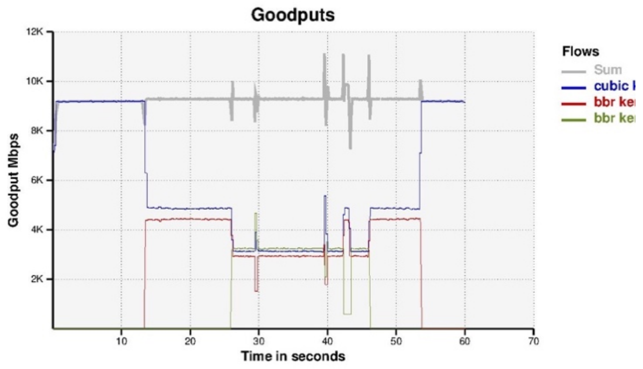

Figure 7C: Cubic vs. 2-BBR with TCP-BPF

Size Fairness and Many Flows

The following experiments consist of 1 streaming flow, 1 10KB back-to-back RPC and 1 or more 1MB back-to-back RPCs. The goals are: (1) see how much bandwidth each flow can get, (2) see what happens as we increase the number of flows. This is still a LAN test.

Figures 8A to 8D show the results for this test. Figure 8A shows the average goodput of the Streaming flows (always 1 per host). The BBR goodput for the streaming is higher than for other algorithms. This may seem good, but it means it is taking bandwidth from the 1MB and 10KB flows (all congestion algorithms fully use the bottleneck link bandwidth). In addition, BBR has much higher retransmissions that any other congestion algorithm.

Figure 8B and 8C show the goodputs of the 1MB and 10KB back-to-back RPCs. As expected, since the streaming flow used so much bandwidth, the goodputs of the 1MB and 10KB RPCs are lower for BBR. NV allows the 10KB flow to achieve the highest goodput by wide margins (greater than 4x factor). That is, small RPC flows benefit from running under NV.

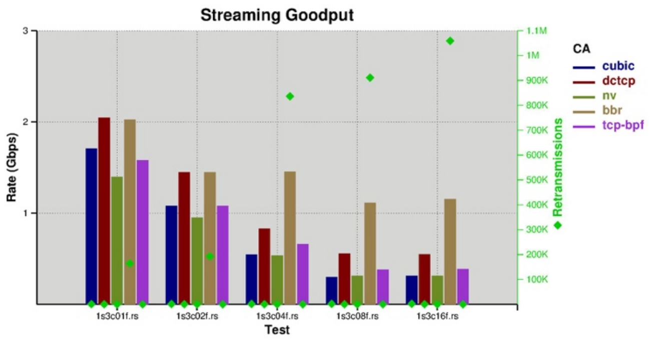

Figure 8A: Streaming goodput with many flows

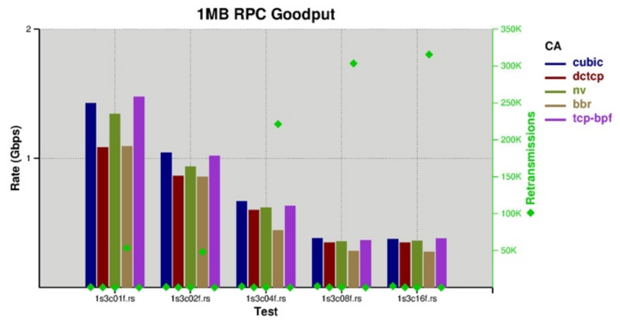

Figure 8B: 1MB Goodputs

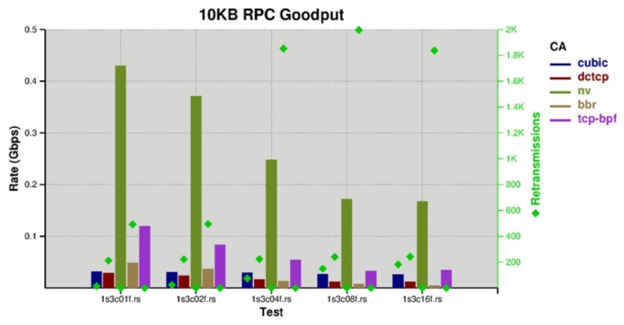

Figure 8C: 10KB Goodputs

Finally, Figure 8D shows the 99 and 99.9 percentile latencies for the 10KB RPCs (in a log scale). NV achieves the lowest latencies and the 99 and 99.9% latencies are the same. TCP-BPF achieves the next lower latencies and they are also the same. BBR has the worst latencies followed by DCTCP. Note that we are seeing retransmissions with DCTCP which we have not seen in the past. This is unexpected and not normal. We are investigating to determine if it is due to a bug in the latest kernel or due to an issue with ECN marking in our switch.

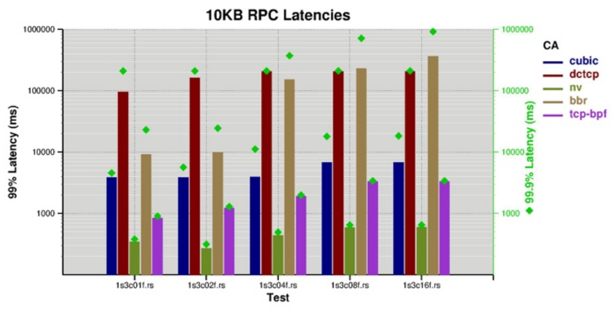

Figure 8D: 10KB 99% and 99% Latencies

10G-10ms Scenario

This scenario introduces a 10ms delay and keeps the 10Gbps rate. It uses a rack switch with its small buffers and NetEm at the receiver to introduce latency. We also include the following congestion algorithms: BIC[10], HighSpeed[7], H-TCP[6], Scalable[9], Westwood[5] and Yeah[1].

2 and 3 Flow, all flows same TCP variant

Figure 9A shows the aggregate goodput (i.e. the sum of the flow goodputs) and the percent retransmissions for many congestion algorithms. The numbers are the averages of 20 runs. First thing to notice is the high rate of retransmissions for BBR; for 2 flows the rate is 0.9%, for 3 flows is 3.4%. All other congestion algorithms have retransmission rates of 0.01% or lower.

The second thing to notice is that many of the congestion algorithms will underutilize the available bandwidth by up to 60% (Westwood). Cubic underutilizes the link by 20% to 12%. The algorithms that best use the bandwidth are BIC, Yeah, Scalable and BBR.

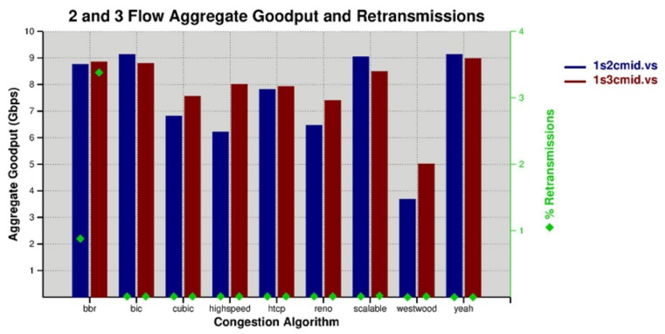

Figure 9A: 2 & 3 Flow Aggregate Goodputs

The 2nd dimension is how fair is the algorithm. That is, if two flows share the bandwidth evenly. Figure 9B shows the average rates of the fastest and slowest flow. Since the 1st flow runs by itself for some time, its rate is higher than that of the 2nd flow. BBR's ratios between slower and faster are close to the ideal; when new flows start the bandwidth is divided evenly between them. We see that BBR, BIC and Yeah are the fairest algorithms.

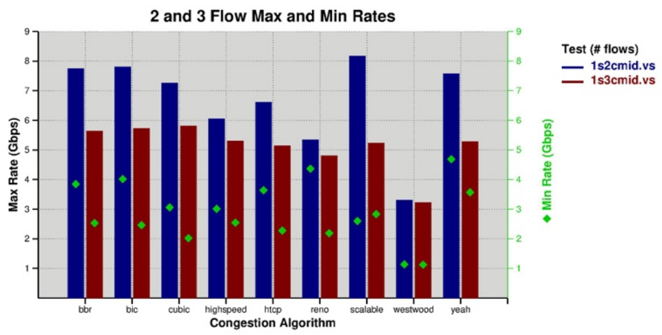

Figure 9B: Min and Max Rates

Each test was ran 10 to 20 times and the various graphs examined to see the algorithm behavior. One interesting result was that in a few cases (10% of 2 flow tests and 20% of 3 flow tests) one of the BBR flows slowed down to 1/100 of the other flows bandwidth for 5 to 40 seconds.

Figure 10A shows the goodputs of 3 BBR flows where the 1st flow suddenly collapses at 42 seconds and remains that way, even after the other flows end. Figure 10B shows the high rate of retransmissions for all flows. Interestingly, we do not see the RTT probing in the 1st graph that should [illegible] the probing among all flows so it occurs at the same time. It does this by using losses as the synchronization signal and when there are too many it just stops probing.

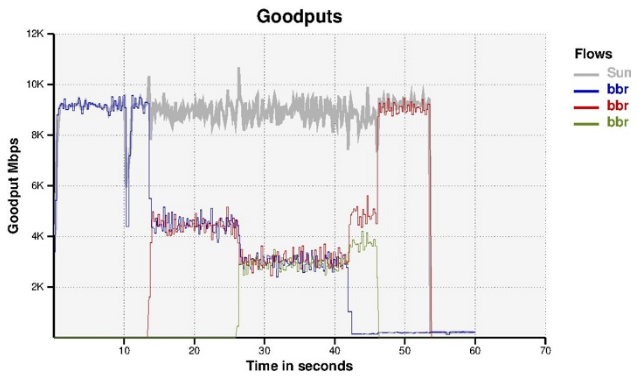

Figure 10A: 3-BBR flows with one collapsing

Figure 11A shows the case where 2 out of 3 BBR flows are collapsed, they never get much throughput. Figure 11B shows the retransmission; surprisingly there are very few as compared to previous case (and most other cases).

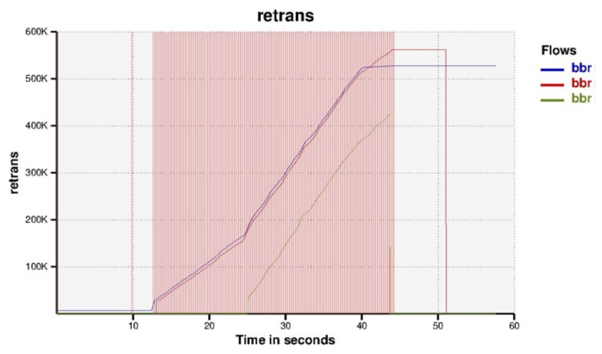

Figure 10B: Retransmissions for 3-BBR flows

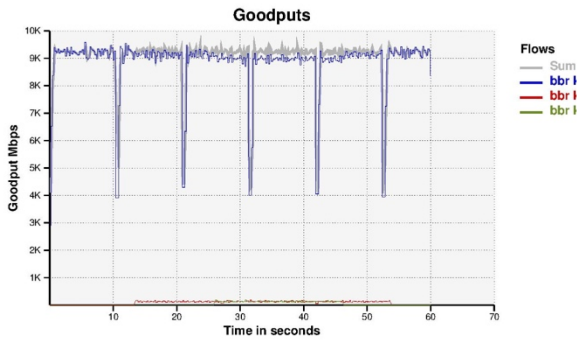

Figure 11A: 3-BBR Flows showing suppressed throughput

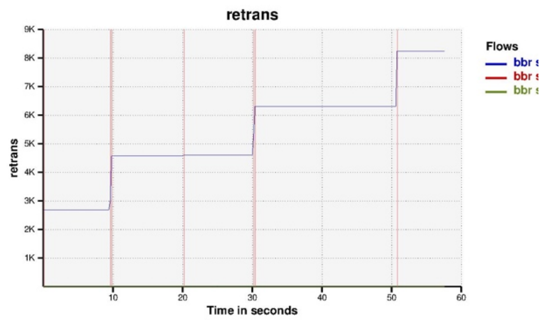

Figure 11B: Retransmissions for 3-BBR flows

However, in most cases 3-flow BBR looks as in figure 12, achieving almost perfect fairness. Its retransmissions look as in those of figure 10B.

In contrast, figures 13A to 13D show the goodputs for Cubic, Reno, Bic and Yeah (not showing for the other algorithms since we know they performed poorly). Both Reno and Cubic take some time to grow their cwnd and to converge their goodputs. In contrast, BIC and Yeah adapt to available bandwidth much faster. Yeah converges to fairness faster than BIC.

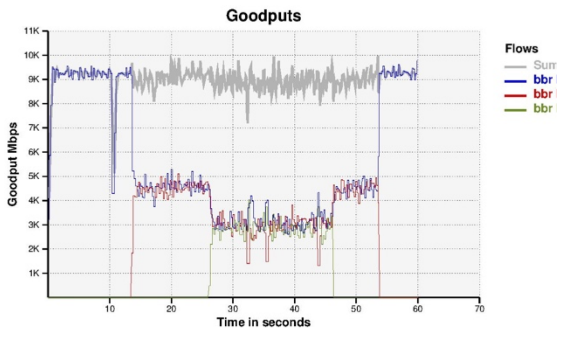

Figure 12: 3-BBR flow common behavior

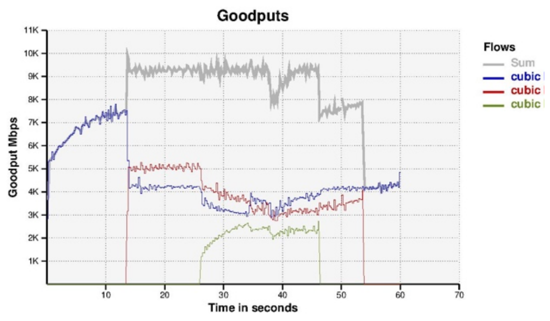

Figure 13A: 3-Cubic flows

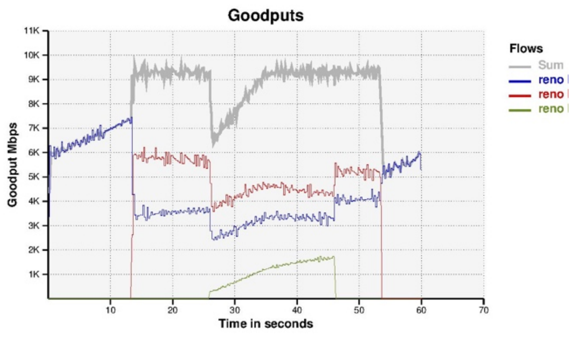

Figure 13B: 3-Reno flows

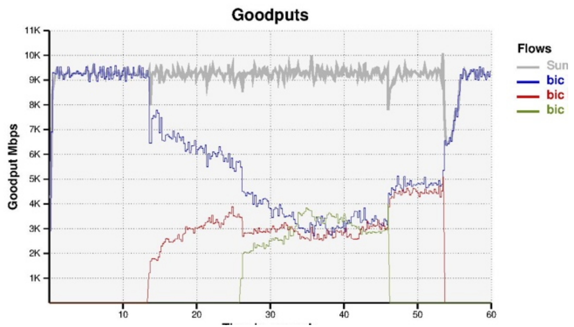

Figure 13C: 3-BIC flows

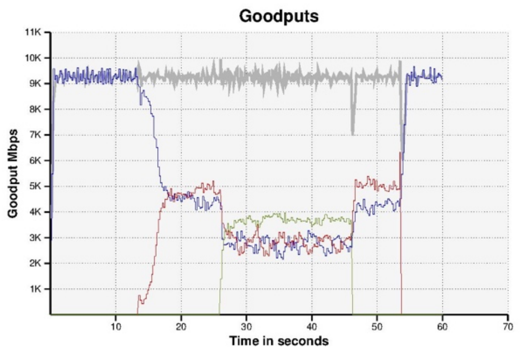

Figure 13D: 3-Yeah flows

3-Flow Cubic vs. Other

Figure 14 shows the aggregate goodputs and retransmissions when Cubic competes with the other algorithms. We only look at BBR, BIC, Reno and Yeah because the others do poorly. The top algorithm shown in the X axis indicates that it started first. From the graph, we see that BBR, BIC and Yeah vs. Cubic achieve the highest aggregate goodput. However, BBR has the most retransmissions at 1.4% while it is less than 0.01% for the others.

The aggregate goodput is lower when Cubic starts first because Cubic is running by itself at the beginning and the end and it is not as efficient as the others. Cubic vs. Yeah does better than Cubic vs. BBR

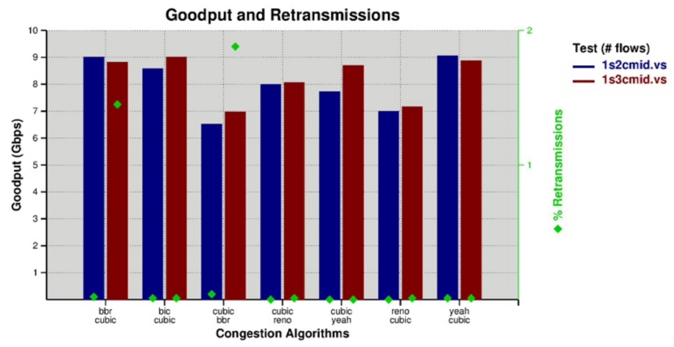

Figure 14: 3-flow versus Goodputs

However, that is not the whole story. We still need to look at fairness. Looking at the graphs shows that Cubic losses against BBR and BIC, Yeah losses against Cubic and Reno and Cubic are even overall.

Size Fairness and Many Flows

We again explore the scenario with many flows of different sizes. Each of the 3 senders is doing one streaming flow, one 1MB back-to-back RPC and 1, 2, 3, 8 and 16 8MB RPCs. Note that we used larger RPCs because the smaller RPCs cannot get much throughput with the larger latency. Figure 15A shows the overall goodput and retransmissions for Cubic, BBR, BIC and Yeah.

Surprisingly the retransmissions (as sown by green diamonds) decrease from 4% to 1.7% as the load increases. BBR has slightly higher throughput at the lowest loads, but it evens up as the load increases.

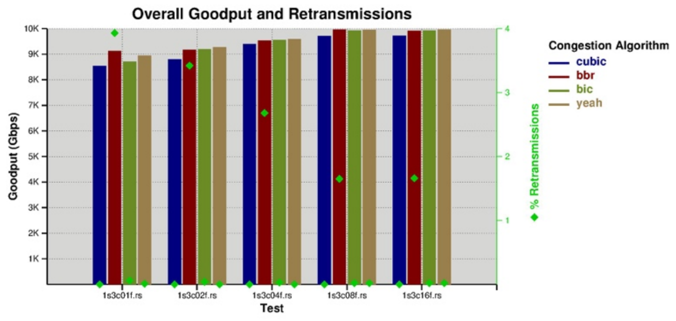

Figure 15A: Overall Goodput and Retransmissions

Figure 15B shows the goodput and retransmissions for the streaming flow. BBR has the highest goodput by far, about twice as fast as the others. However, since all the congestion algorithms are using all of the available bandwidth, this means the streaming flow is taking bandwidth away from the other flows. This is seen in figures 15C and 15D which show the goodput for the 8MB and 1MB flows. Now the BBR flows are getting much lower goodputs (especially the 1MB RPCs). This is an indication that, at least in this scenario, BBR is size unfair. That is, fatter flows get ore of the bandwidth.

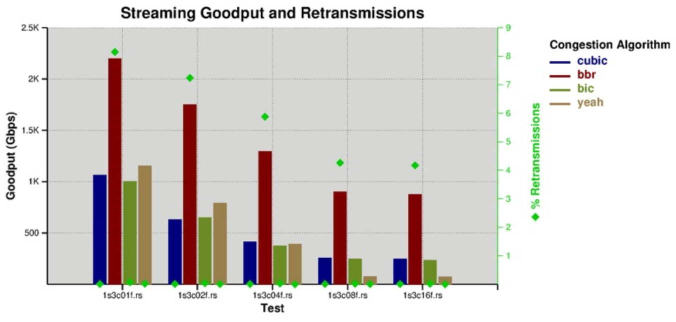

Figure 15B: Goodput of streaming flows

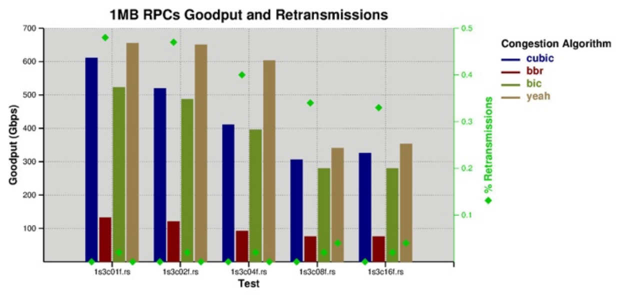

Figure 15D: Goodput of 1MB RPCs

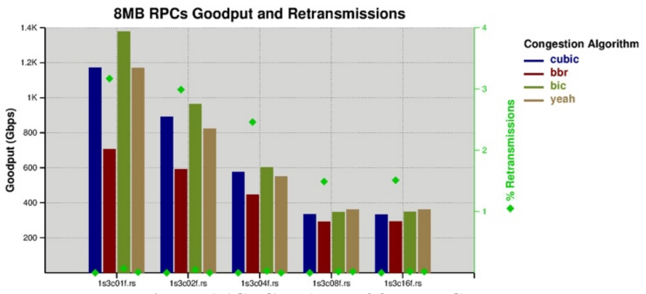

Figure 15C: Goodput of 8MB RCPs

Finally, Figure 15E shows the 99 and 99.9 percentile latencies of the 1MB RPCs. As expected they are much higher for BBR. They are almost 10x larger.

All of this means that when there are flows of different sizes the smaller ones will lose against the larger ones when using BBR. There are two reasons for this unfairness. The first one is that with BBR, larger flows get much higher cwnds increasing the RTT. Since the RPCs take at least 1 RTT, their goodput put decreases. The second reason is that the higher number of retransmissions lead to more RTOs.

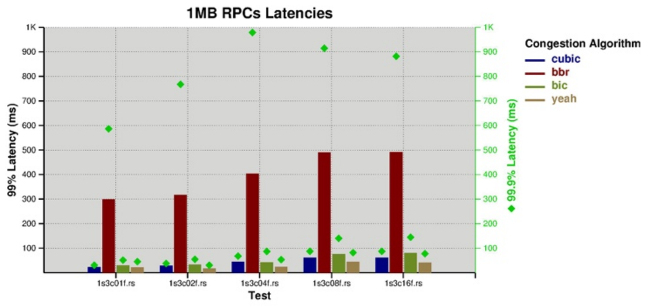

Figure 15E: 99 and 99.9 % latencies of 1MB RPCs

40ms Scenarios

These scenarios use a 40ms delay at 10 Mbps rates. As before, the delay is introduced to the ACKs by the receiver. The buffer sizes at the bottleneck ranged from 8X to half the BDP in order to analyze the effect of buffer size on the congestion algorithms. For these parameters, 40ms RTT and 10 Mbps, 16 buffers is half of BDP, 32 is BPD, 64 is 2xBDP, etc.

Size Fairness and Many Flows

Figure 16A shows the overall goodput and retransmissions vs. buffer size at the bottleneck when we only have 1 flow per host. That is, 1 host is streaming, another host is doing back-to-back 1MB RPCs and the 3rd host is doing back-to-back 10KB RPCs.

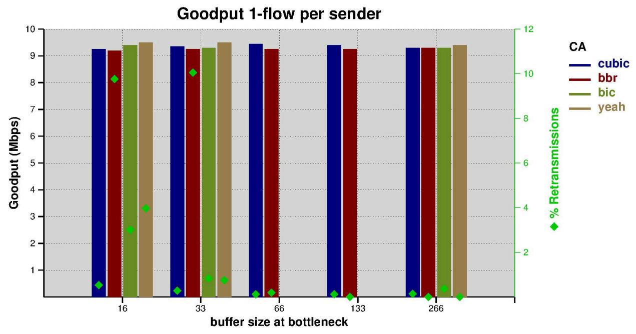

Figure 16A: Overall goodput and retransmissions, 1 flow per host

All congestion algorithms have similar goodput, but BBR has much higher retransmissions with the small buffer sizes. This is especially true when the buffering is 1xBDP.

Figure 16B shows the goodput and 99 percentile latency of the 10KB flow. The 10KB goodput and 99% latency are worst for BBR with buffer sizes less or equal to 1xBDP as compared to the other congestion algorithms. However, it starts doing better as the amount of buffering increases.

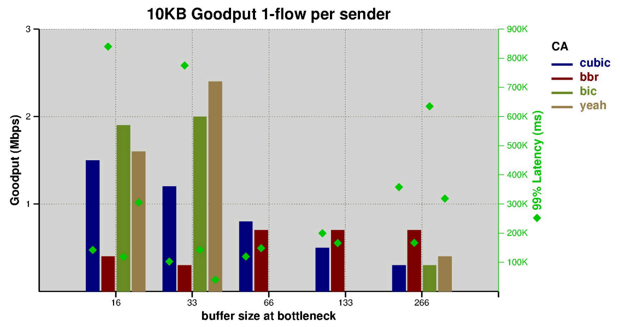

Figure 16B: Goodput and latency of 10KB RPC, 1 flow per host

Figures 17A and 17B show similar graphs for the case where each host has 3 flows: streaming, 1MB RPCs and 10KB RCP. Now BBR is worst in terms of oval retransmissions and 10KB latency for all buffer sizes. This seems to indicate that, at least for the parameters of this experiment, the amount of buffering BBR needs to perform well is a function of the level of congestion.

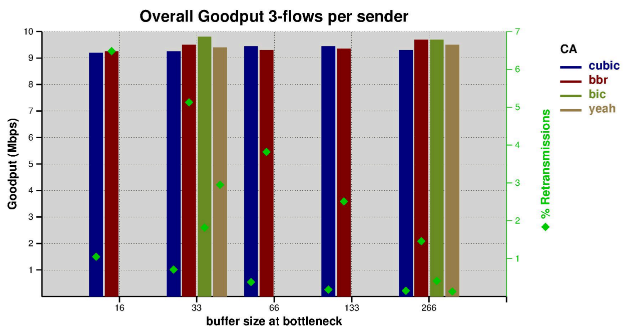

Figure 17A: Overall goodput and retransmissions, 3 flows per host

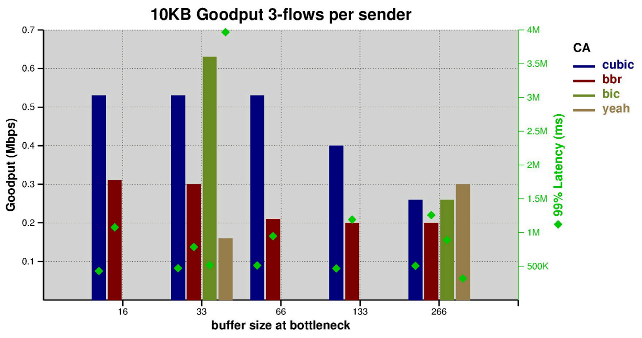

Figure 17B: 10KB goodput and 99% latency, 3 flows per host

Conclusions

As we have shown, no congestion algorithm is better than the others under in all conditions. However, it is important that we understand their weakness and strengths before they are deployed widely.

In particular, here are some conclusions per congestion algorithm.

- DCTCP. There were issues in the experiments that we have not seen in the past. An investigation is ongoing to determine whether this is an issue with our experimental setup for DCTCP (how marking was done) or an issue with the latest version of DCTCP.
- NV. Performed well in DC experiments but more analysis is needed with more complex workloads to insure there are no issues. As mentioned earlier, NV is currently not suited for non-DC traffic.
- TCP-BPF. Using TCP-BPF to limit cwnd sizes does improve DC performance when there are not too many flows, but its advantage decreases as the number of active flows increases.
- Cubic. Perform badly in the 10G-10ms experiments. This was surprising since Cubic is supposed to do well with larger RTTs. It performed much worse than BIC, its predecessor and this was unexpected. Not clear if this has always been the case or it is the result of modifications to Cubic.
- BIC. Did well with larger RTTs and larger bandwidths.
- BBR. Mixed bag. Did well on some scenarios but was very unfair, even to itself, on other scenarios. It seems that BBR needs large buffering to perform well, but the amount of buffering may be a function of the load.

References

1. A. Baiocchi, A. Castellani, and F. Vacirca. YeAH-TCP: Yet Another Highspeed TCP. *In proc. International Workshop on Protocols for Fast Long-Distance Networks*, Marina del Rey, California, USA, February 2007.
2. Brakmo, L. S., Peterson, L.L. 1995. TCP Vegas: end-to-end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications*13(8): 1465-1480.
3. Brakmo, L. 2010. TCP-NV: Congestion Avoidance for Data Centers. *Linux Plumbers Conference*, Massachusetts, U.S.A.
4. Brakmo, L. 2017. Network Testing with Netesto. *Netdev 2.1 Technical Conference*, Montreal, Canada.
5. C. Casetti, M. Gerla, S. Mascolo, M. Sansadidi, R. Wang, "TCP Westwood: end-to-end congestion control for wired/wireless networks", *Wireless Netw. J.*, vol. 8, pp. 467-479, 2002.
6. D. Leith and R. Shorten. H-TCP Protocol for High-Speed Long Distance Networks. *In proc. International Workshop on Protocols for Fast Long-Distance Networks*, Argonne, Illinois, USA, February 2004.
7. Floyd, S. 2003. HighSpeed TCP for Large Congestion Windows. *IETF RFC 3649*.
8. Jacobson, V. 1988. Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review* 18(4): 314-329.
9. Kelly, t. "Scalable TCP: improving performance in highspeed wide area networks", *Comput. Commun. Rev.*, vol. 32, no. 2, Apr. 2003.
10. Lisong Xu, Khaled Harfoush, and Injong Rhee, Binary Increase Congestion Control for Fast, Long Distance Networks, *Infocom*, *IEEE*, 2004
11. Mario Hock, Roland Bless, Martina Zitterbart, "Experimental Evaluation of BBR Congestion Control", IEEE ICNP 2017, October 2017
12. Mohammad Alizadeh , Albert Greenberg , David A. Maltz , Jitendra Padhye , Parveen Patel , Balaji Prabhakar , Sudipta Sengupta , Murari Sridharan,

Data center TCP (DCTCP), *Proceedings of the ACM SIGCOMM 2010 conference*, August 30-September 03, 2010, New Delhi, India

13. Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, Van Jacobson, “BBR: Congestion Based Congestion Control,” *Communications of the ACM*, Vol. 60 No. 2, Pages 58-66. (https://cacm.acm.org/magazines/2017/2/212428-bbr-congestion-based-congestion-control/fulltext)
14. Sangtae Ha, Injong Rhee and Lisong Xu, CUBIC: A New TCP-Friendly High-Speed TCP Variant, *ACM SIGOPS Operating System Review*, Volume 42, Issue 5, July 2008, Page(s):64-74, 2008.
15. S. Hemminger “Network Emulation with NetEm” *Linux Conf Au 2005*. http://developer.osdl.org/shemminger/ LCA2005 paper.pdf