TCP Congestion Algorithms In Datacenters

Large Systems · Project Report

Luc Gommans, Rick van Gorp

December 24, 2017

Abstract

In datacenters, multiple tenants share a finite amount of bandwidth. When an uplink is saturated, packets that overflow available buffer space will be dropped and have to be retransmitted by the sender. When one tenant chooses to retransmit faster than others, they have an advantage as they will make it though more quickly. TCP is a connection-oriented protocol which employs congestion control algorithms. These algorithms aim to divide the available bandwidth equally between network flows.

Some congestion control algorithms are more aggressive than others, and it has been reported that some algorithms cause others to back off excessively. Tenants may, perhaps inadvertently, cause unequal bandwidth distributions during times of congestion.

In this research, we compare five algorithms under different network conditions to assess how they influence each other. We found that only CTCP (Windows) backs off excessively; all other algorithms behave fairly. The results for CTCP are anomalous compared to all other algorithms and we conclude that for this algorithm, future work is needed for a reliable assessment.

1 Introduction

Multitentant datacenters aim to provide services with consistent performance and reliability. When customers send a lot of data in a short amount of time, networks might become saturated or even congested, and bandwidth needs to be distributed fairly among tenants.

Congestion occurs when a buffer on the path to the recipient is full and packets have to be dropped. TCP uses a congestion control algorithm to avoid congestion in the network and distribute bandwidth fairly among users. Several algorithms are available to perform this function. Each has a different behaviour depending on the network's characteristics.

This report includes a performance and behaviour analysis of different TCP congestion control algorithms in a (simulated) multitenant datacenter network. Mario Hock et al. describe in a paper that BBR shows unfair behaviour when compared to the CUBIC congestion control algorithm[9]. Therefore the fairness between congestion control algorithms will also be included in the analysis. Based on the results, we will recommend datacenters on measures to take to keep providing a reliable environment for customers when multiple congestion control algorithms are active in the network, for example using Quality of Service or prohibiting certain TCP congestion control algorithms. The main research question is: *How can a datacenter provide a reliable and performing environment for customers when multiple TCP congestion control algorithms are used in the network?*

In the next section, we will discuss related work. Section 3 describes our methods: how we selected algorithms to test, how we tested their performance, and how we chose network conditions to simulate. Section 4 describes the results of our research: the algorithms we chose to test are described in 4.1, a theoretical analysis of those is described in 4.2, and experimental results can be found in 4.3. We discuss our results in section 5, mention possible future work in section 6, and finally draw conclusions in section 7.

2 Related work

In 2010, the algorithm DCTCP was described by Mohammad Alizadeh et al.[1] and published as RFC8257[5] in 2017. DCTCP is optimised for datacenters and provides a high-burst tolerance, low latency, and high throughput when the datacenter has a small part of the buffer available[5].

In 2016 another algorithm was proposed: BBR. This is a TCP congestion algorithm created by Google, which achieves higher bandwidths and lower latencies compared to other TCP congestion methods[3]. A comparison of BBR with CUBIC[9] shows that the BBR node pushes the CUBIC node away in bandwidth when using small buffers. The BBR node gets more bandwidth allocated than the CUBIC node.

The TCP congestion control algorithm BBR is discussed by Neal Cardwell et al.[3] The article contains a performance test of the BBR algorithm and a comparison between the BBR algorithm and the CUBIC algorithm, where a noticeable difference in bandwidth allocation is shown when using small buffers.

A. Esterhuizen and A.E. Krzesinski[4] analyse the performance of multiple TCP congestion control algorithms, specifically: Reno, BIC, CUBIC, HighSpeed TCP, TCP-Hybla, TCP-Illinois, TCP Low Priority, TCP-Vegas, TCP-Westwood, TCP-YeAH, and Scalable TCP.

3 Method

During this research, performance comparison tests of various TCP congestion control algorithms will be performed. The most common TCP congestion algorithms will be determined through desk research. The tests will be conducted in an isolated environment. Based on the results of those tests, recommendations will be given related to the measures a datacenter has to take to keep the traffic equally allocated over customers.

3.1 Selecting TCP congestion algorithms

In order to select the most common TCP congestion algorithms desk research will be performed. This desk research will show which algorithms are used more often than other algorithms by looking for the default algorithms in Operating Systems. Based on those results, a list of TCP congestion algorithms will be composed.

3.2 Comparing Performance

We will analyse one common scenario: multiple servers of different owners inside a multitenant datacenter, sends data to clients outside the datacenter. This could be users downloading software, the bursts while streaming a video or movie, or large resources on a web page.

We will create a bottleneck on the path to the user to see how different congestion control algorithms handle this. The connection will be full-duplex and have symmetrical speeds, to avoid any congestion in sending acknowledgements. Because the server is sending the large volume of data and the client is responding with the much smaller acknowledgements, the server will be the one applying congestion control. Therefore we vary the algorithm settings of the server and not the client. The client will always be a GNU/Linux machine using CUBIC, but we assume (due to the aforementioned reasons) that this does not matter.

The bottleneck is created by having one receiving host, listening on different ports, and two transmitting hosts. Each network interface is 1 gigabit per second, thus the servers will have to limit their speed in order to avoid flooding the client. This setup is shown in figure 1.

We use three physical hosts as servers: two running Ubuntu 17.04 and one running Microsoft Windows 10. In each experiment, only two hosts will be active. The Windows system is used because there is no working CTCP implementation—the default algorithm in Windows since Vista—available for a modern Linux kernel.

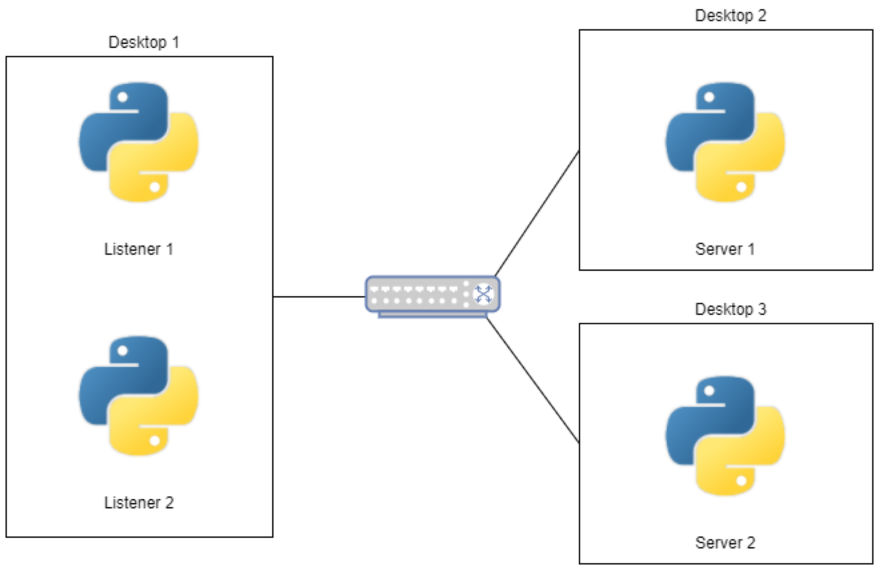

Figure 1: Test setup - Scenario 1

We will test the congestion algorithm for fairness in sharing bandwidth with other TCP connections using other congestion algorithms. In order to test the fairness, the bandwidth usage per second per algorithm is measured. If two different algorithms use similar amounts of bandwidth, they can be said to be fair relative to each other.

The tests conducted with the algorithms have two environment variables: packet loss and delay. Those environment variables are simulated using `netem`, which is a network emulation functionality in several Linux distributions[6]. `netem` is applied to the incoming side of the interfaces of the clients to emulate the same packet loss and delay. Windows, which is tested as a server, does not support `netem`. Therefore the packet loss and delay are applied to the incoming interfaces of the clients. Packet loss and delay are chosen as the behaviour of most TCP algorithms depends on those variables[3][5][8][12][14]. The values for the environment variables are chosen based on measurements performed by Verizon[13].

Each test is run at least twice. When results are conflicting or unexpected, we ran a test more often and on different hardware.

3.2.1 Determining the Packet loss and Delay

The data set of Verizon[13] is used to calculate the packet loss and delay used in this research. The data set includes global information and shows averages per month over the year 2017. The packet loss is calculated by taking the average of the Verizon Business Packet Delivery Statistics for Country Specific Metrics for each country. Those averages were inverted by subtracting the packet delivery percentage from 100, to show the actual packet loss in percentages and not the packet delivery. Based on the ascending sorted values a bar chart was created. This bar chart is shown in figure 2

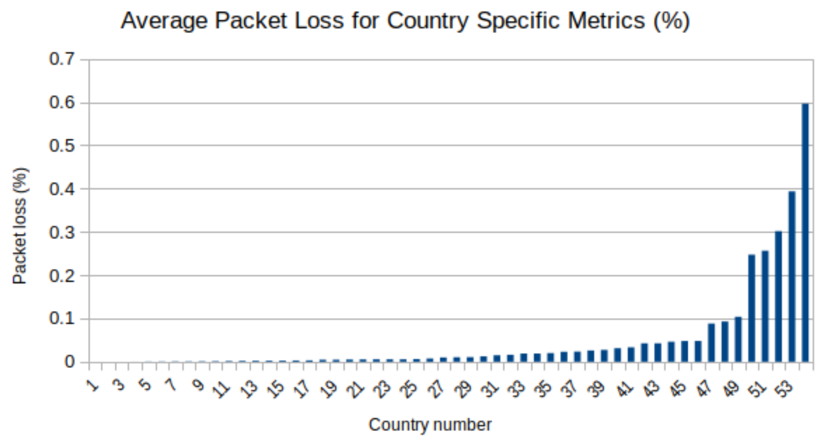

Figure 2: Average Packet Loss for Country Specific Metrics

The country numbers in figure 2 are only used to show the distribution of the average packet loss. The maximum average packet loss shown in figure 2 equals 0.6% and the minimum equals 0%. However, to account for situations where more packet loss occurs than usual, the double value of the maximum is used: 1.2%. The values for packet loss in this comparison set-up were determined based on the distribution shown in figure 2 and the manually set maximum of 1.2%. The values are shown in table 1.

Packet loss
0%.
0.01%
0.1%
0.6%
1.2%

Table 1: Packet loss percentages used for the comparison setup

The delay is calculated by taking the average of the Verizon Business Latency Statistics for Country Specific Metrics for each country. Based on the ascending sorted values a bar chart was created. This chart is shown in figure 3.

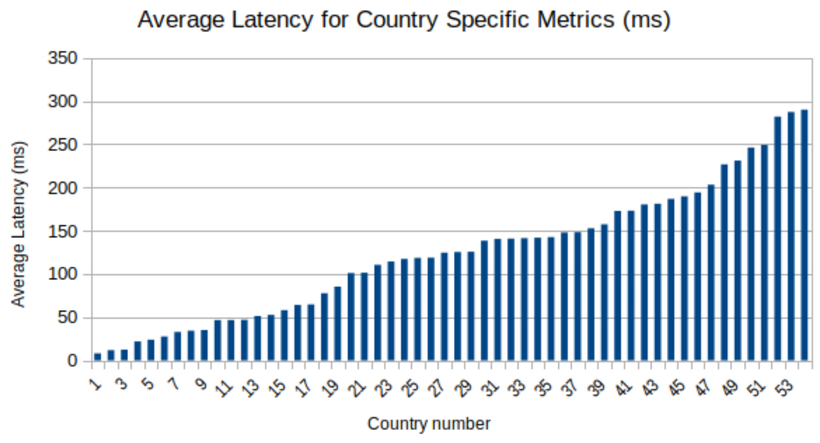

Figure 3: Average Latency for Country Specific Metrics

The country numbers in figure 3 are only used to show the distribution of the average delays. The maximum delay shown in figure 3 equals 290ms and the minimum equals 8ms (both rounded). The values for delay in this comparison set-up were determined based on the distribution shown in figure 3. The values are shown in table 2.

Delay
8ms
64ms
120ms
176ms
232ms
290ms

Table 2: Packet loss percentages used for the comparison setup

3.2.2 Performing time-based automated tests

The tests will be performed using scripts written in Python[7]. All nodes run a client, which performs actions as instructed by the controller. The controller is responsible for controlling the tests: synchronising time among nodes, setting the TCP congestion algorithm of a node, and requesting one node to open a listening port and instructing another to start sending at a given time. Nodes report their measurements back to the controller.

3.2.3 Testing hardware

The specifications of the hardware used for the tests are shown in tables 3 and 4.

Table 3: Node configurations

Hardware	**Specification**
Model	Dell Optiplex 7010
CPU	Intel(R) Core(TM) i5-3570S CPU @ 3.10GHz
Memory	12GB
NIC	Intel 82579LM Gigabit Network Connection (rev 04)
NIC Driver	e100e 3.2.6-k

Table 4: Switch configuration

Hardware	**Specification**
Model	3Com OfficeConnect Gigabit Switch 16
Ports	16x Gigabit Ethernet

3.3 Traffic Management

Desk research will be conducted to gather information related to traffic management. The goal is to find a mechanism that is capable of allocating bandwidth equally between customers of a datacenter. This is done by analysing the results from the algorithm comparison and the initial desk research on traffic management.

4 Results

In this section we will first elaborate on which algorithms we looked into, next we look at their internal workings, and finally we show the results of our experiments.

4.1 TCP Congestion Algorithms

The most common TCP congestion algorithms are CTCP, CUBIC and BIC. CTCP is the default congestion algorithm in Microsoft Windows Server 2008 and newer[10]. CUBIC is the default congestion algorithm in Linux distributions with kernel versions 2.6.19 and higher[10]. BIC was the default congestion algorithm in earlier Linux distributions from kernel version 2.6.8 until 2.6.19[11][10]. DCTCP and BBR will also be tested as those protocols are new and upcoming—they are included in Linux since 4.9 (December 2016)[2]. DCTCP is specifically designed for datacenters[5] and BBR is a new algorithm meant for general use[3].

4.2 Theoretical Algorithm Characteristics

The BBR algorithm depends on parameters $BtlBw$ and RT_{prop}[3]. Those parameters are estimates defined by the BBR algorithm. $BtlBw$ specifies the bandwidth at the slowest link in each direction and RT_{prop} is the round-trip propagation time. $RT_{prop} = RTT - queuingdelay - processingdelay$, which results in the minimum amount of time for round trip propagation in case there are no queuing or processing delays. Using those two parameters, the Bandwidth Delay Product (BDP) is calculated, which is the maximum possible amount of data being sent in a network: $BDP = BtlBw * RT_{prop}$. As long as the amount of data in transit is less than the BDP and $BtlBw$, the delivery rate is increased. If the amount of data in transit is equal to $BtlBw$ the delivery rate can not go up anymore. The RTT can never be lower than RT_{prop}.

The CTCP algorithm is a combination of a loss-based and delay-based congestion protocol[12]. For the loss-based component, the variable conventional congestion window, $cwnd$, was introduced. For the delay-based component, the variable delay window, $dwnd$, was introduced. The TCP sending window is determined from: $win = min(cwnd + dwnd, awnd)$, where $awnd$ is the window advertised by the receiver. For $cwnd$ on the arrival of an ACK: $cwnd = cwnd + 1/win$. The $dwnd$ variable is initially set to zero and is only set when the congestion avoidance is active. $dwnd$ is calculated differently for several scenarios. If the network path is underutilised: $dwnd(t+1) = dwnd(t) + \alpha * dwnd(t)^k - 1$. When the network path is congested: $dwnd(t+1) = dwnd(t) - \eta * diff$. When packet loss occurs: $dwnd(t+1) = dwnd(t)(1-\beta) - cwnd/2$. k, α, η and β are tunable parameters for optimisation of the algorithm. η defines the rapidness of reduction of the window when congestion is detected. β is the factor used for the multiplicative decrease after loss is detected. $diff$ is calculated from the expected throughput and the actual throughput, which are again calculated from the window size and Round Trip Time (RTT).

The DCTCP algorithm[1] consists of three components:

1. Simple Marking at the Switch. Based on the marking threshold K a packet is marked with the CE flag. This only occurs when the queue occupancy is greater than K at arrival time. The marking threshold K specifies the minimum value of queue occupancy.

2. ECN-Echo at receiver. The receiver running the DCTCP algorithm ACKs every packet and sets the ECN-Echo flag only if the packet was marked with CE. The ECN-Echo flag is used to notify the sender of congestion in the network.

3. Controller at the sender. The sender has an estimate of the fraction of packets that are marked with CE. This is based on the actual fraction of packets that were marked, the weight given to the new estimate compared to the past weight. This results in α. α is used to indicate the congestion on a network with a maximum of value 1, indicating a congested network, and a minimum value of 0, indicating no congestion in the network. DCTCP cuts the window size using α: $cwnd = cwnd * (1 - \alpha/2)$. This results in a low queue length, but ensuring high throughput.

The BIC algorithm controls its windows based on the size of windows[14]. It consists of two components:

1. Binary search increase. This algorithm computes the midpoint between the maximum window size W_{max} and current minimum window size W_{min}. If packet loss occurs, W_{max} is set to the midpoint. This process repeats until W_{max} and W_{min} are smaller than the minimum increment S_{min}.

2. Additive increase runs as an addition to the binary search increase. If $W_{max} - midpoint > S_{max}$, the window size is increased by S_{max} until $W_{max} - midpoint < S_{max}$. S_{max} is the preset maximum increment.

In case the current window size becomes greater than W_{max} a slow start strategy is performed to determine a new W_{max}: $W_{max} + \alpha * S_{min}$, where α is a multiplier. The multiplier is increased by one until $S_{min} * \alpha >= S_{max}$ or when losses occur. In the case of loss a new value for W_{max} will be set.

The CUBIC algorithm is a successor of the BIC algorithm[8]. In case of a loss event W_{max}, which is the maximum window size, is registered. This is followed by a multiplicative decrease of the congestion window by factor β. If an ACK in congestion avoidance is received and the current window size *cwnd* is less than W_{max}, the algorithm uses the concave profile to increase the window size according to the formula: $cwnd = \frac{W(t+RTT)-cwnd}{cwnd}$. If *cwnd* is larger than W_{max}, the convex profile is used to increase the window size according to the same formula as the concave profile. In both formulas t is the time elapsed from the last window reduction.

4.3 Experiments' Results

In our results, we see that all algorithms clearly respond to the simulated network circumstances, but not very much to each other. Most algorithms are fair towards all other algorithms and will use their half of the connection. See figures 4 and 5. In figure 4 we see that in the first test run, the increase of BIC is more aggressive than CUBIC's. This is related to the difference in increase functions used by both TCP congestion algorithms. In figure 5 we also see that the increase of BIC is more aggressive than DCTCP's. BIC's increase function is characterised to be more aggressive and that can be seen in the graph as the increase line is more steep. The differences in those increase functions, however, do not show any implications on the fairness of the algorithms as the bandwidth allocation is near to equal for both algorithms.

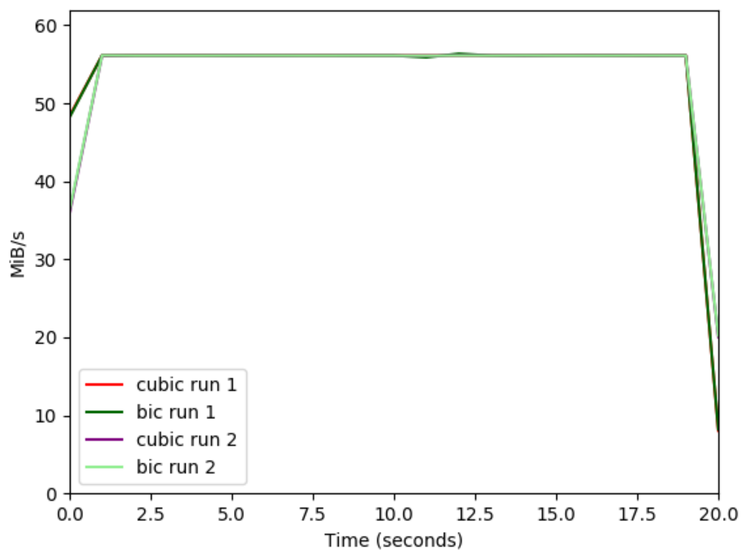

Figure 4: CUBIC vs BIC with 8ms latency and 0% loss

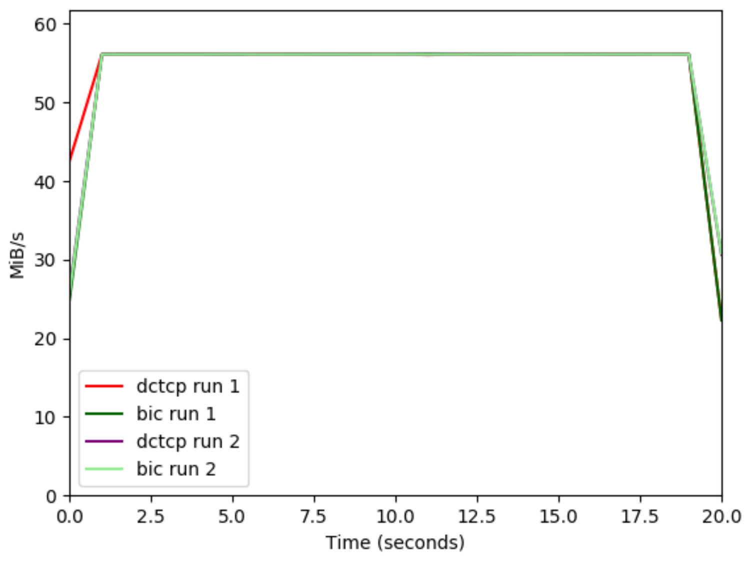

Figure 5: DCTCP vs BIC with 8ms latency and 0% loss

CTCP is the only algorithm that responds excessively to others. Alone, it reaches the full gigabit speed; but if any other algorithm runs at the same time (i.e. a normal test is performed), CTCP's speed consistently drops. It then reaches around 57 megabits (±6.8MiB/s), while the other algorithms reach at least 800 with good network conditions (8 milliseconds latency, 0% loss). See figure 6 for an example of BIC vs. CTCP.

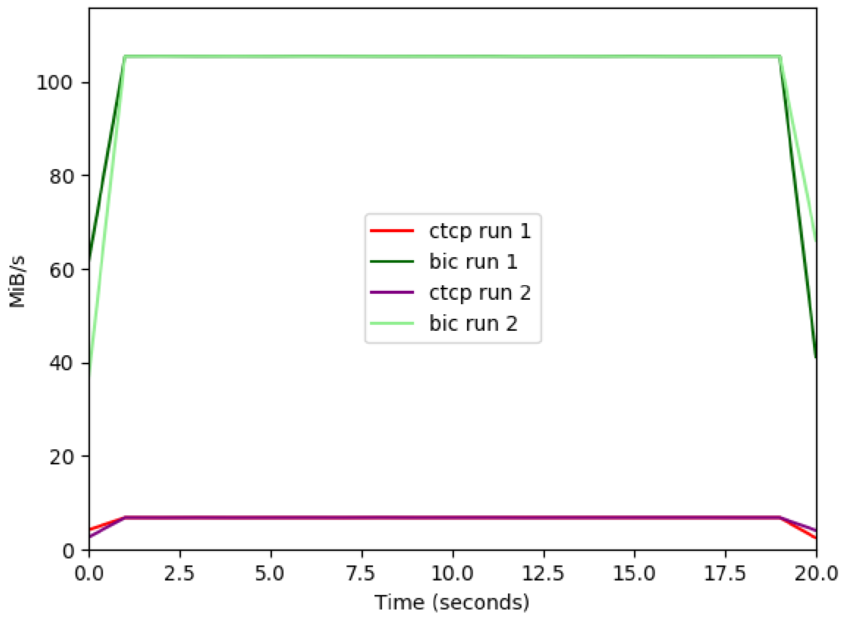

Figure 6: CTCP vs. BIC with 8ms latency and 0% loss

Additionally, CTCP also lowers its speed in response to the simulated network circumstances: with 1.2% packet loss and 8 milliseconds latency, it will reach around 25 megabits, and with 290 milliseconds latency and no loss, it reaches around 1.5 megabits. In neither case does it reach the original 55 megabits.

Because of the exceptional results for CTCP, we performed the test again on different hardware. This yielded the same result. To verify that it is CTCP and not our software that performs differently on Windows, we setup a small experiment using `iperf`. The setup was the same as our original experimental setup with one sender on two hosts and two receivers on one host, only we manually started `iperf` bandwidth tests rather than doing it in an automated fashion with Python. In this instance, Windows performed normally and did not back off exceptionally when introducing another contender.

BBR is unique in that it barely responds to packet loss but only to delay: even with 1.2% loss, as long as the delay is low (8 milliseconds in our case), it reaches 850±20 megabits per second. A comparison between BBR and CUBIC is shown in figure 7. We see in figure 7 that, compared to CUBIC, the bandwidth usage by the BBR data transfer is higher when 1.2% packet loss is introduced. However, the link speed of 1Gbps is not fully utilized as is in the paper written by Mario Hock et al.[9]. We also do not see the CUBIC data transfer responding to the BBR data transfer by reducing its speed in any way. In another test, where CUBIC is compared to BIC with the same parameters, we see CUBIC utilizing the same amount of bandwidth as in the BBR-CUBIC comparison. This is shown in figure 8. In figure 9 we show a comparison of a CUBIC data transfer with a BBR data transfer with parameters 8ms latency and 0% packet loss. This time the total link speed is utilized and we see no unfairness between CUBIC and BBR. We therefore see no signs in our results of BBR being the cause of the CUBIC data transfer being slower. In our results other algorithms also reach much lower speeds (24 megabits is not uncommon), but we see no signs of BBR being the cause of that. In tests with BBR nor CTCP, speeds are not higher than when BBR is one of the algorithms being tested.

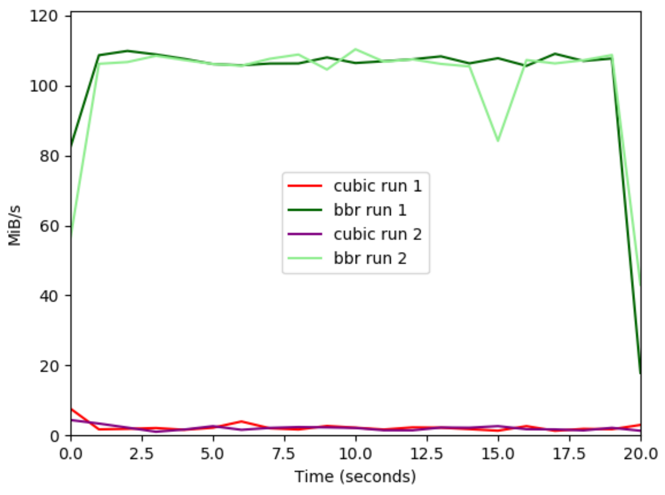

Figure 7: CUBIC vs. BBR with 8ms latency and 1.2% loss

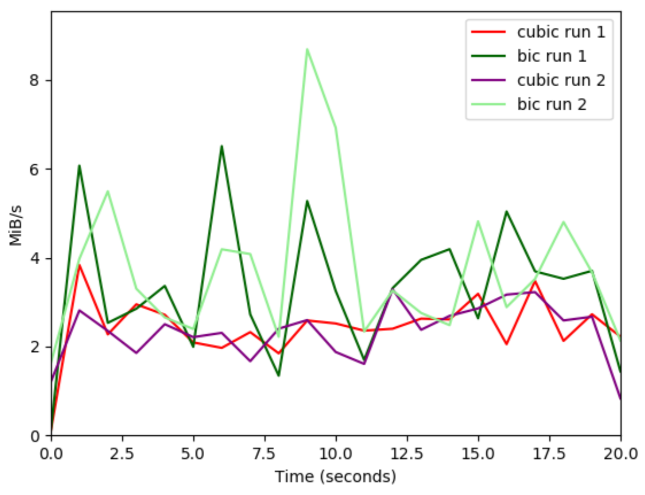

Figure 8: CUBIC vs BIC with 8ms latency and 1.2% loss

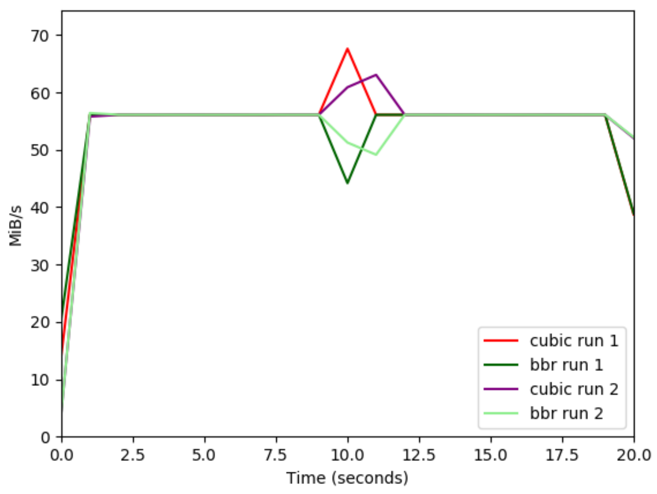

Figure 9: CUBIC vs. BBR with 8ms latency and 0% loss

The full set of results is available online[7].

4.4 Traffic Management

Our results show that traffic management is currently not necessary. However, a tenant might purposefully install an unfair algorithm. In this section we discuss different possible approaches a datacenter could take.

We propose two possible solutions to manage traffic from a multitenant datacenter to customers when unfair TCP congestion control algorithms are used: traffic shaping based monitoring, and detection of unaccepted TCP congestion algorithms in the network.

4.4.1 Traffic shaping based Monitoring

With the use of a hypothetical congestion control algorithm which is unfair towards other flows, the bandwidth allocation to one customer would be higher than to other customers. When a TCP flow shows an aggressive increase and keeps pushing data over the network, other (fair) algorithms will reduce the transmission rate as packet loss occurs. One important condition for this unfair congestion control algorithm is that it should not be aggressive to the extent of causing a denial of service.

To mitigate this, the bandwidth usage per server could be monitored, prioritising the traffic sent by some servers. The monitor uses traffic shaping to control the priority of the traffic per server:

1. The monitor measures periodically whether the bandwidth in the link is fully utilised. If this is the case, the next step will be performed. If not, it continues monitoring the link utilisation.
2. The monitor calculates periodically, per server, whether its bandwidth usage deviates more than α from the average bandwidth usage. α is the deviated bandwidth and is set according to an acceptable deviation value. The following formula is used to calculate the standard deviation: $s_i = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N-1}}$. N is the amount of active servers on the link. $\overline{x}$ is the average amount of bandwidth for all active servers on the link. x_i is the bandwidth of the current server and s_i shows the standard deviation in bandwidth for server i. If s_i is a negative value, no priority change will be applied as the bandwidth usage is lower than average for the server. If $s_i <= \alpha$, then no priority change will be applied. If $s_i > \alpha$, the priority will be changed to a lower value.
3. The monitor provides information of the servers where the priority will be set to a lower value to the traffic shaping controller.
4. The traffic shaping controller sets the priority for the servers to a lower value, so that the traffic coming from the servers that use α more than average bandwidth has less preference than traffic closer to the average bandwidth and lower.
5. The monitor periodically checks if the bandwidth in the link is still fully utilised. If this is not the case, the traffic shaping policy is reversed again by setting the

priority to the default value. If this is the case, the periodical calculations of server bandwidth usage are continued.

4.4.2 Detection of unaccepted TCP congestion control algorithms

Peng Yang et al.[15] describe a method to remotely identify used TCP congestion avoidance algorithms. The paper states that they were able to identify all default TCP algorithms, but some non-default algorithms could not be identified. The tests performed are limited to remote web servers, so it requires adaption to implement such a method in a generic manner. If this method is deemed reliable, it can be useful to detect whether an unfair algorithm is used as it will not match any of the default algorithms used. Based on that, the traffic of the unacceptable TCP congestion control algorithms can be shaped by lowering their priority.

5 Discussion

The CTCP measurements were done on a Windows 10 machine because there is no up-to-date implementation of CTCP for Linux. All other measurements were done on machines running Ubuntu 17.04. This might have introduced issues while measuring the CTCP algorithm versus any other congestion algorithm. According to our measurements, all other algorithms are fair in sharing bandwidth with each other; only CTCP consistently and excessively reduced its bandwidth allocation.

Without any concurrent flows, Windows does reach the full link speed (1Gbps). On different hardware, it showed the same, recessive behaviour. We think this might be caused by the way Python socket operations are handled in Windows. When repeating the test with `iperf`, which is written in C, this behaviour no longer occurs. Even when using the Python equivalent of `while(true){ socket.send(data); }`, the bandwidth allocation did not change. We theorise that the difference is caused by something in the `send` call. An alternative theory is that Microsoft Windows for workstations have a different implementation than Microsoft Windows Server systems.

BBR is the best-performing algorithm in our set, because it barely responds to packet loss. However, we were unable to draw the same conclusions to our results as the conclusions written in the paper by Mario Hock et al[9]. We used multiple values to simulate packet loss as well as delay, which simulate having small buffers. BBR is a TCP congestion control protocol that reacts to the minimum RTT in combination with the Bottleneck Bandwidth. We have tested multiple delay values that appear in the real world. The delay values are directly related to the minimum RTT possible on the link. As mentioned in the results, we see no hard evidence that this causes other algorithms to reduce their speed further, but there are slight indications. For example in the test of DCTCP versus BBR, DCTCP reaches around 0.2 to 0.6 megabytes per second. In the test of DCTCP versus BIC, this is 0.3 to 0.7 megabytes per second. The difference is large nor statistically significant, but it warrants looking into. On the scale of 0.7 megabytes per second, a tenth of a megabyte does make a difference.

6 Future work

Running a mixed environment of Linux and Windows introduced issues, at least in combination with Python as benchmark system. We expect that in Windows, the Python send call on sockets is implemented differently than in Linux, but this requires more testing. The tests for CTCP should be performed again when the reason for the discrepancy is known and can be mitigated. The results should also be verified using a Microsoft Windows Server instance.

Our tests were done using three nodes in a switched network. To make sure our results hold in practice, future work could introduce a routed network with many more nodes.

In this paper we have proposed a monitoring solution, which checks for the deviation between the server bandwidth on a link and the average bandwidth used by servers on a link. This solution requires performance testing, such that it is capable of running in a datacenter and not causing delay. Also an α value has to be determined that works in most environments, without ruling out servers that are not running unfair TCP congestion protocols.

BBR performs better in a congested network than any other tested algorithm. While we did not see clear signs of other algorithms responding to BBR's higher bandwidth share, this should be tested further to be certain. We suggest starting a transfer with one algorithm under the desired network circumstances, determine its bandwidth, and then start a parallel flow using BBR. If the bandwidth of the original flow does not decrease, BBR is indeed fair towards this algorithm.

7 Conclusion

Our main research question is: *How can a datacenter provide a reliable and performing environment for customers when multiple TCP congestion control algorithms are used in the network?*

For the algorithms BIC, BBR, CUBIC and DCTCP, we conclude that, based on the setup we have used, they allocate bandwidth fairly and do not suppress each other. There is no impact on the service for customers when using those algorithms mixed in one environment.

CTCP, the default algorithm for Microsoft Windows, shows inconclusive results and we can not draw any conclusions with regards to this algorithm.

In our results, BBR shows better performance than any other algorithm when packet loss occurs. Finally, DCTCP is only suitable when both the server and client are in the same datacenter: there is a considerable ramp-up time when congestion occurs. Within datacenters DCTCP works well, because without congestion it keeps queue lengths to a minimum.

References

[1] Mohammad Alizadeh et al. *Data Center TCP (DCTCP)*. URL: https://people.csail.mit.edu/alizadeh/papers/dctcp-sigcomm10.pdf (visited on 11/16/2017).

[2] Diego Calleja. *Linux 4.9*. 2017. URL: https://kernelnewbies.org/Linux_4.9#head-1e42ba62fbcb6f54176e9e31c69bf22be06aab0f.

[3] Neal Cardwell et al. "BBR: Congestion-Based Congestion Control". In: *ACM Queue* 14, September-October (2016), pp. 20 –53. URL: http://queue.acm.org/detail.cfm?id=3022184.

[4] A. Esterhuizen and A.E. Krzesinski. "TCP Congestion Control Comparison". In: (2012). URL: http://www.satnac.org.za/proceedings/2012/papers/2.Core_Network_Technologies/15.pdf (visited on 11/16/2017).

[5] Internet Engineering Task Force. *Data Center TCP (DCTCP): TCP Congestion Control for Data Centers*. URL: https://tools.ietf.org/html/rfc8257 (visited on 11/16/2017).

[6] The Linux Foundation. *Netem*. 2016. URL: https://wiki.linuxfoundation.org/networking/netem (visited on 11/30/2017).

[7] Luc Gommans Rick van Gorp. *TCP Congestion Control in Datacenters*. 2017. URL: https://github.com/lgommans/lsproj (visited on 12/14/2017).

[8] Sangtae Ha, Injong Rhee, and Lison Xu. *CUBIC: A New TCP-Friendly High-Speed TCP Variant*. 2016. URL: https://www.researchgate.net/profile/Sangtae_Ha/publication/220623913_CUBIC_a_new_TCP-friendly_high-speed_TCP_variant/links/574b20e608ae5f7899ba14ec/CUBIC-a-new-TCP-friendly-high-speed-TCP-variant.pdf.

[9] Mario Hock, Roland Bless, and Martina Zitterbart. *Experimental Evaluation of BBR Congestion Control*. URL: http://doc.tm.kit.edu/2017-kit-icnp-bbr-authors-copy.pdf (visited on 11/16/2017).

[10] Adam Krajewski. *TCP loss sensitivity analysis*. 2016. URL: https://indico.cern.ch/event/558754/contributions/2387602/attachments/1380046/2097516/ATCF2-20161130-TCP-algorithms-comparison.pdf (visited on 12/01/2017).

[11] Yee-Ting Li and Dough Leith. *BicTCP Implementation in Linux Kernels*. 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.146.4069&rep=rep1&type=pdf (visited on 12/01/2017).

[12] Kun Tan Jingmin Song, Qian Zhang, and Murari Sridharan. *Compound TCP: A Scalable and TCP-Friendly Congestion Control for High-speed Networks*. 2007. URL: http://www.dcs.gla.ac.uk/~lewis/CTCP.pdf (visited on 12/11/2017).

[13] Verizon. *IP Latency Statistics*. 2017. URL: http://www.verizonenterprise.com/about/network/latency/ (visited on 11/30/2017).

[14] Lisong Xu, Khaled Harfoush, and Injong Rhee. *Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks*. 2004. URL: http://infocom2004.ieee-infocom.org/Papers/52_4.PDF.

[15] Peng Yang et al. *TCP Congestion Avoidance Algorithm Identification*. 2011. URL: `http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1159&context=cseconfwork`.