Experiences Evaluating DCTCP

Lawrence Brakmo, Boris Burkov, Greg Leclercq, Murat Mugan

Facebook
Menlo Park, USA

Abstract

In this paper we describe our preliminary experiences evaluating DCTCP[12] for use in our data centers. Our testing corroborates results from other groups indicating that DCTCP is very effective in decreasing packet losses and retransmissions. In addition, DCTCP increases fairness between RPC flows of different sizes.

We also discovered issues in our NIC's firmware and the DCTCP implementation in the Linux kernel that affected DCTCP performance.

Finally, we discovered that DCTCP can increase CPU utilization under some conditions.

Keywords

TCP, Linux, congestion algorithms, DCTCP

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. Work on this problem has been going on for more than 30 years and still continues to this date.

Congestion control was added to TCP in 1988 by Van Jacobson[8] to deal with congestion collapse, a network condition where most of the bandwidth ends up being wasted by packets that are later retransmitted. Since then many new congestion control algorithms have been developed and added to Linux's repertoire of TCP flavors [1,2,3,5,6,7,9,10,12,13,14].

Most of these congestion algorithms depend on packet losses to detect network congestion. As a result, they fill queues along bottleneck links resulting in standing queues (non-dissipating queues) increasing latency.

These standing queues also introduce unfairness between RPCs of different lengths. When the RTT increases from tens of micro-seconds to 1 millisecond, a 10KB RPC can only do 80Mbps at best, while a 1MB RPC could theoretically go up to 8Gbps.

Furthermore, loss-based congestion control algorithms increase tail latency as a result of delays in detecting packet losses. Rather than preventing congestion, loss-based congestion control algorithms need to periodically create packet losses to detect that the available bandwidth is fully utilized.

In contrast, congestion avoidance algorithms try to detect congestion before losses occur. Most congestion avoidance algorithms try to detect congestion by detecting growing queues, an early stage of congestion. Most, like TCP-Vegas[2] and BBR[13,11] use the RTT to detect queue growth. Others, like DCTCP[12] use explicit congestion signals from switches and routers.

ECN (Explicit Congestion Notification) is a framework allowing switches to share congestion signals to senders and receivers. IP headers use 2 bits for sharing ECN information. If none of the bits are set, then the flow does not support ECN and switches will not mark the packets. When only one bit is set, then the flow supports ECN signals and no congestion has been encountered. Finally, when both bits are set, then the flow supports ECN and the packet has encountered congestion.

Congestion is detected through queue sizes. In the simplest case, when a packet arrives, if the queue used to temporarily store the packet is larger than some threshold, then the packet is marked as having experienced congestion. This signal then arrives at the receiver, and the receiver then needs to notify the sender so it can adjust its rate appropriately. For TCP flows, the signal is sent back on the TCP header of the ACK packet.

The standard (pre-DCTCP) response of TCP is to reduce its congestion window (cwnd) as if a packet had been dropped. That is, Reno would reduce its cwnd by 50%. This is a very aggressive reduction that can lead to link underutilization in some networks. In addition, the standard response does not differentiate between transient (very short lived) and standing (long lived) congestion. For example, congestion events (queues larger than a given threshold) that last less than an RTT would still result in reducing cwnd.

In contrast, DCTCP employs a mechanism where the cwnd is reduced proportionally to the level of congestion. It achieves this by tracking the percentage of bytes per RTT that encounter congestion and reducing the cwnd proportionally. Thus if 100% of the bytes encounter congestion, then DCTCP would reduce its cwnd by 50%. Whereas if only 50% of the packets encounter congestion, then the cwnd would only be reduced by 25%.

In practice, DCTCP uses a moving average in order to deal with transient congestion. For example, if 100% of the bytes in an RTT encounter congestion, but there was no congestion in previous RTTs, then the cwnd would only be reduced by 1/32 instead of ½.

Overview

In order to determine the suitability of deploying DCTCP in our data centers, we ran various tests. We started with simple 3 to 4 server intra-rack tests and finished with full rack tests consisting of 4 to 6 racks.

Intra-Rack Tests

For the 3 to 4 server within rack tests, we used Netesto[4] for the tests. Netesto (Network Test Toolkit) is a framework for running network tests which simplifies the process of running complex tests as well as collecting test metrics and analyzing the test results.

These tests consisted of 2 or 3 senders and one receiver. Figure 1shows the topology for these tests. The traffic consisted of a mix of 1MB and 10KB RPCs. The primary value of these tests is to verify the expected behavior of DCTCP: reduction of packet losses, decrease of tail latency and increased fairness between 1MB and 10KB RPCs.

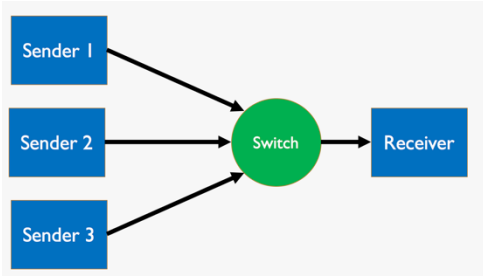

Figure 1: Topology for intra-rack tests

Inter-Rack Tests

The topology of the inter-rack tests is shown in Figure 2. It consisted of 3 full racks of storage nodes and 3 full racks of worked nodes. The worker nodes read data from the storage nodes and do a small amount of processing. In addition, there is some amount of traffic between the worker nodes and also between the storage nodes. While the applications where actual production applications, the load was artificial in order to fully saturate the links between the work and storage racks.

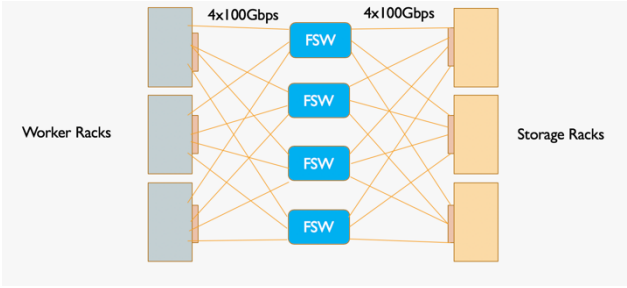

Figure 2: Topology for inter-rack tests

Intra-Rack Tests Results

As mentioned earlier, these tests consisted of multiple 1MB and 10KB concurrent RPCs. Through this testing we uncovered the following issues:

- Unfairness between senders, regardless of congestion control used
- Unfairness between flows when using ECN
- High tail latencies when using DCTCP

Causes and fixes of these issues are discussed in the following sub-sections.

Unfairness Between Senders, Regardless of CC

We noted unfairness when 3 senders send to a fourth one (see Figure 1). Senders 1 and 2 would each get 25% of the bandwidth, while Sender 3 would get 50% of the bandwidth. The cause turned out to be due to a design issue on the switch. The switch has 2 output buffers, with half of the input ports using one buffer and the other half using the other buffer. The switch round-robins between buffers when sending packets out of the output ports.

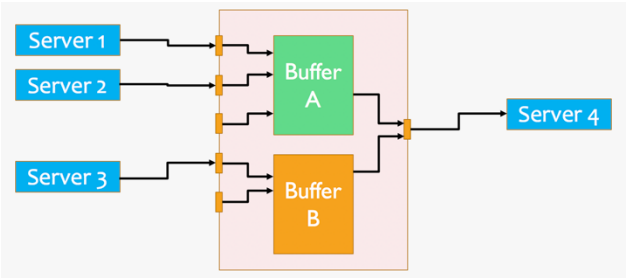

Figure 3: Switch architecture

As Figure 3 shows, Servers 1 and 2 were using buffer A, while Server 3 was using buffer B causing the unfairness. To remove this issue, we only chose servers using buffer A for our testing.

Unfairness Between Flows When Using ECN

With only two flows, the first flow would get 23Gbps and the second one would only get 0.5Gbps of bandwidth (we were using a 25Gbps NIC). We wrote a tool to analyze pcaps that could show per RTT metrics such as throughput, duration, number of packets sent as well as per packet information. This showed that the RTTs of the second flow were bimodal: either around 60us or 1.2ms and that the cwnd was small, less than 20.

Figure 4 shows some of the output from one test run. The numbers on the second column indicate the flow number (2) followed by the RTT number (separated by a period). Note that an RTT starts when a packet is sent (either the first data packet or the first after a previous RTT ends) and ends when its ACK is received. The third column shows the time when the RTT finished (starting with time 0 when the SYN packet is sent) and its duration. The fourth column shows the number of bytes that were sent during the RTT (usually

cwnd) and the final column shows the rate achieved during the RTT (out bytes divided by RTT).

```
RTT  2.1     0.020255( 1.3ms)   out: 14.3KB rate: 85.83 Mbps
RTT  2.2     0.021586( 1.3ms)   out: 28.6KB rate:173.88 Mbps
RTT  2.3     0.022900(  40us)   out: 30.0KB rate:  5.85 Gbps
RTT  2.4     0.022941( 1.3ms)   out:  2.9KB rate: 17.35 Mbps
RTT  2.5     0.024258(  57us)   out: 31.4KB rate:  4.41 Gbps
RTT  2.6     0.024315( 1.3ms)   out:  1.4KB rate:  8.72 Mbps
RTT  2.7     0.025625(  76us)   out: 32.8KB rate:  3.46 Gbps
RTT  2.8     0.025701( 1.2ms)   out: 32.8KB rate:210.71 Mbps
RTT  2.9     0.026948(  85us)   out: 35.7KB rate:  3.36 Gbps
RTT  2.10    0.027033( 1.3ms)   out: 30.0KB rate:187.72 Mbps
RTT  2.11    0.028311( 1.4ms)   out: 38.6KB rate:222.87 Mbps
RTT  2.12    0.029695(  67us)   out: 41.4KB rate:  4.94 Gbps
```

Figure 4: RTT Analysis of Slow Flow

After further analysis, it turned out the cause was due to coalescing values used by a new feature of the NIC firmware that were not shown in our version of the ethtool. We fixed the issue by updating the firmware and disabling the feature.

High Tail Latencies When Using DCTCP

After fixing the previous issues and comparing Cubic and DCTCP behavior we noticed that the 99 and 99.9% latencies were much higher for DCTCP. These are shown in Figure 5.

	Cubic Latencies		DCTCP Latencies	
	99%	99.9%	99%	99.9%
1-MB RPCs	2.6ms	5.5ms	43ms	208ms
10-KB RPCs	1.1ms	1.3ms	53ms	212ms

Figure 5: High DCTCP Tail Latencies

After analyzing pcaps using our tool and looking at the Linux network stack source code we discovered the following issues:

- RTOs caused by the receive sending a duplicate ACK instead of ACKing the last (and only) packet sent.
- The receiver delaying ACKs when the sender had a cwnd of 1, so the sender paused for the duration of the delayed ACK (40ms)

It turned out that there some old bugs in the DCTCP and ECN handling code that were triggered by a patch in 2015 (I.e. this issue has been around for about 3 years). These bugs are now fixed thanks to patches from Yuchung Cheng, Neal Cardwell and me.

With these fixes, the tail latencies of DCTCP now look much better and are shown in Figure 6. Note that the 10KB latencies are much better (5 to 10) and that the reason that the 1MB latencies are now larger than Cubic's is because DCTCP is allowing the 10KB RPC to get much better throughput (so there is less bandwidth for the 1MB RPC). That is, it is something positive not a negative. DCTCP solved the unfairness between RPC sizes that occurs with Cubic and is caused by Cubic creating standing queues which increase RTTs.

	Cubic Latencies		DCTCP (fixed) Latencies	
	99%	99.9%	99%	99.9%
1-MB RPCs	2.6ms	5.5ms	5.8ms	6.9ms
10-KB RPCs	1.1ms	1.3ms	146us	203us

Figure 6: Fixed DCTCP Tail Latencies

Inter-Rack Tests Results

These tests used 4 to 6 fully populated racks (see Figure 2) and consisted of worker nodes reading from the storage nodes using an artificial load in order to increase utilization of the network. We started by using the full 3 racks of workers which could only saturate up to 70% of the FSW links going to the worker racks. These results are shown in Figure 7.

	Cubic	DCTCP
FSW to Worker Max Link Util %	69.9	69.8
FSW Discards (bits)	89M	236K (0.3%)
Worker rack discards (bits)	417M	0
Storage Retransmits	0.020	0.000
Worker Retransmits	0.173	0.078
Storage CPU (%)	X	X
Worker CPU (%)	Y	Y + 1%
Storage ECN CE Marked (%)		6.5
Worker ECN CE Marked (%)		12.8

Figure 7: Results when using 3 worker racks

DCTCP has fewer discards (300x) and retransmissions with only a small increase in CPU usage (1%) on the worker nodes as compared to Cubic. Note that the maximum link utilization (from the FSW switches to the worker racks) was only 70% on average.

In order to increase network load, we only used 2 worker racks in the next tests. This increased maximum link utilization to 99% and switch discards under Cubic increased by almost 500x. In contrast, DCTCP has 1000x less discards and 500x less retransmissions. However, CPU utilization increased on both the worker (14%) and storage nodes (4%). These results are shown in Figure 8 below.

	Cubic	DCTCP
FSW to Worker Max Link Util %	99.1	98.7
FSW Discards (bits)	160B	157M (0.1%)
Worker rack discards (bits)	2.2B	0
Storage Retransmits	0.590	0.001
Worker Retransmits	0.376	0.035
Storage CPU (%)	X	X + 14%
Worker CPU (%)	Y	Y + 4%
Storage ECN CE Marked (%)		5.5
Worker ECN CE Marked (%)		63.7

Figure 8: Results when using 2 worker racks

Note that 63.7% of the packets received by the worker nodes are ECN marked as having experienced congestion.

For the final tests we only used one worker rack, further increasing network congestion. This resulted in small increases in Cubic discards and retransmissions, but much

larger increases in DCTCP discards and retransmissions. Even though DCTCP is very effective in decreasing discards, there is only so much it can do if the network load is too high. These results are shown in Figure 9 below.

	Cubic	DCTCP
FSW to Worker Max Link Util %	99.9	98.1
FSW Discards (bits)	235B	19B (8.1%)
Worker rack discards (bits)	1.1B	0
Storage Retransmits	1.020	0.125
Worker Retransmits	0.620	0.125
Storage CPU (%)	X	X + 10%
Worker CPU (%)	Y	Y + 3%
Storage ECN CE Marked (%)		18
Worker ECN CE Marked (%)		73

Figure 9: Results when using only 1 worker rack

CPU overheads decreased while percent of ECN congestion marked packets increased to 73%.

The likely cause of CPU increase is the decrease in packet coalescing due to smaller congestion windows (cwnd) at the senders (i.e. smaller TSO/GSO packets) and at the receivers due to not being to coalesce packets with ECN congestion markings with packets without ECN congestion markings. In addition, the DCTCP receiver also sends more ACK packets further increasing the load on the sender.

It is not clear how much of an issue this will be on production traffic since it is better behaved than the artificial load we used in these tests.

Conclusions

Smaller, intra-rack, tests are very important in helping discover and fix any issues may occur due to software bugs or experimental conditions. Our tests also indicate that we, the Linux kernel community, have we increase our test coverage. It is surprising that we had bugs that affected DCTCP tail latencies for more than 3 years.

Intra-rack and inter-rack tests show conclusively that DCTCP is very good at reducing packet discards and improving tail latencies. They also show that DCTCP improves the RPC latencies and goodput of smaller RPCs as compared to Cubic.

Futher work is needed to determine if CPU overheads will be an issue with production workloads. If so, we must decrease these overheads.

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.