Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2016

S. Bensley
Microsoft
L. Eggert
NetApp
D. Thaler
P. Balasubramanian
Microsoft
G. Judd
Morgan Stanley

July 3, 2015

Microsoft's Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters
draft-bensley-tcpm-dctcp-04

Abstract

This memo describes Datacenter TCP (DCTCP), an improvement to TCP congestion control for datacenter traffic. DCTCP enhances Explicit Congestion Notification (ECN) processing to estimate the fraction of bytes that encounter congestion, rather than simply detecting that some congestion has occurred. DCTCP then scales the TCP congestion window based on this estimate. This method achieves high burst tolerance, low latency, and high throughput with shallow-buffered switches.

Status of This Memo

Table of Contents

1. Introduction

Large datacenters necessarily need a large number of network switches to interconnect the servers in the datacenter. Therefore, a datacenter can greatly reduce its capital expenditure by leveraging low cost switches. However, low cost switches tend to have limited queue capacities and thus are more susceptible to packet loss due to congestion.

Network traffic in the datacenter is often a mix of short and long flows, where the short flows require low latency and the long flows require high throughput. Datacenters also experience incast bursts,

where many endpoints send traffic to a single server at the same time. For example, this is a natural consequence of MapReduce algorithms. The worker nodes complete at approximately the same time, and all reply to the master node concurrently.

These factors place some conflicting demands on the queue occupancy of a switch:

- The queue must be short enough that it does not impose excessive latency on short flows.

- The queue must be long enough to buffer sufficient data for the long flows to saturate the path bandwidth.

- The queue must be short enough to absorb incast bursts without excessive packet loss.

Standard TCP congestion control [RFC5681] relies on segment loss to detect congestion. This does not meet the demands described above. First, the short flows will start to experience unacceptable latencies before packet loss occurs. Second, by the time TCP congestion control kicks in on the sender, most of the incast burst has already been dropped.

[RFC3168] describes a mechanism for using Explicit Congestion Notification (ECN) from the switch for early detection of congestion, rather than waiting for segment loss to occur. However, this method only detects the presence of congestion, not the extent. In the presence of mild congestion, it reduces the TCP congestion window too aggressively and unnecessarily affects the throughput of long flows.

Datacenter TCP (DCTCP) enhances ECN processing to estimate the fraction of bytes that encounter congestion, rather than simply detecting that some congestion has occurred. DCTCP then scales the TCP congestion window based on this estimate. This method achieves high burst tolerance, low latency, and high throughput with shallow-buffered switches.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. DCTCP Algorithm

There are three components involved in the DCTCP algorithm:

- o The switch (or other intermediate device on the network) detects congestion and sets the Congestion Encountered (CE) codepoint in the IP header.

- o The receiver echoes the congestion information back to the sender using the ECN-Echo (ECE) flag in the TCP header.

- o The sender reacts to the congestion indication by reducing the TCP congestion window (cwnd).

3.1. Marking Congestion on the Switch

The switch indicates congestion to the end nodes by setting the CE codepoint in the IP header as specified in Section 5 of [RFC3168]. For example, the switch may be configured with a congestion threshold. When a packet arrives at the switch and its queue length is greater than the congestion threshold, the switch sets the CE codepoint in the packet. For example, Section 3.4 of [DCTCP10] suggests threshold marking with a threshold K > (RTT * C)/7, where C is the sending rate in packets per second. However, the actual algorithm for marking congestion is an implementation detail of the switch and will generally not be known to the sender and receiver. Therefore, sender and receiver MUST NOT assume that a particular marking algorithm is implemented by the switching fabric.

3.2. Echoing Congestion Information on the Receiver

According to Section 6.1.3 of [RFC3168], the receiver sets the ECE flag if any of the packets being acknowledged had the CE code point set. The receiver then continues to set the ECE flag until it receives a packet with the Congestion Window Reduced (CWR) flag set. However, the DCTCP algorithm requires more detailed congestion information. In particular, the sender must be able to determine the number of sent bytes that encountered congestion. Thus, the scheme described in [RFC3168] does not suffice.

One possible solution is to ACK every packet and set the ECE flag in the ACK if and only if the CE code point was set in the packet being acknowledged. However, this prevents the use of delayed ACKs, which are an important performance optimization in datacenters.

Instead, DCTCP introduces a new Boolean TCP state variable, DCTCP Congestion Encountered (DCTCP.CE), which is initialized to false and stored in the Transmission Control Block (TCB). When sending an ACK,

the ECE flag MUST be set if and only if DCTCP.CE is true. When receiving packets, the CE codepoint MUST be processed as follows:

1. If the CE codepoint is set and DCTCP.CE is false, send an ACK for any previously unacknowledged packets and set DCTCP.CE to true.

2. If the CE codepoint is not set and DCTCP.CE is true, send an ACK for any previously unacknowledged packets and set DCTCP.CE to false.

3. Otherwise, ignore the CE codepoint.

3.3. Processing Congestion Indications on the Sender

The sender estimates the fraction of sent bytes that encountered congestion. The current estimate is stored in a new TCP state variable, DCTCP.Alpha, which is initialized to 1 and MUST be updated as follows:

```
DCTCP.Alpha = DCTCP.Alpha * (1 - g) + g * M
```

where

- o g is the estimation gain, a real number between 0 and 1. The selection of g is left to the implementation. See Section 4 for further considerations.

- o M is the fraction of sent bytes that encountered congestion during the previous observation window, where the observation window is chosen to be approximately the Round Trip Time (RTT). In particular, an observation window ends when all the sent bytes in flight at the beginning of the window have been acknowledged.

In order to update DCTCP.Alpha, the TCP state variables defined in [RFC0793] are used, and three additional TCP state variables are introduced:

- o DCTCP.WindowEnd: The TCP sequence number threshold for beginning a new observation window; initialized to SND.UNA.

- o DCTCP.BytesSent: The number of bytes sent during the current observation window; initialized to zero.

- o DCTCP.BytesMarked: The number of bytes sent during the current observation window that encountered congestion; initialized to zero.

The congestion estimator on the sender MUST process acceptable ACKs as follows:

1. Compute the bytes acknowledged (TCP SACK options [RFC2018] are ignored):

   BytesAcked = SEG.ACK - SND.UNA

2. Update the bytes sent:

   DCTCP.BytesSent += BytesAcked

3. If the ECE flag is set, update the bytes marked:

   DCTCP.BytesMarked += BytesAcked

4. If the sequence number is less than or equal to DCTCP.WindowEnd, then stop processing. Otherwise, the end of the observation window was reached, so proceed to update the congestion estimate as follows:

5. Compute the congestion level for the current observation window:

   M = DCTCP.BytesMarked / DCTCP.BytesSent

6. Update the congestion estimate:

   DCTCP.Alpha = DCTCP.Alpha * (1 - g) + g * M

7. Determine the end of the next observation window:

   DCTCP.WindowEnd = SND.NXT

8. Reset the byte counters:

   DCTCP.BytesSent = DCTCP.BytesMarked = 0

Rather than always halving the congestion window as described in [RFC3168], when the sender receives an indication of congestion, the sender MUST update cwnd as follows:

cwnd = cwnd * (1 - DCTCP.Alpha / 2)

Thus, when no sent byte experienced congestion, DCTCP.Alpha equals zero, and cwnd is left unchanged. When all sent bytes experienced congestion, DCTCP.Alpha equals one, and cwnd is reduced by half. Lower levels of congestion will result in correspondingly smaller reductions to cwnd.

Just as specified in [RFC3168], TCP should not react to congestion indications more than once every window of data. The setting of the "Congestion Window Reduced" (CWR) bit is also exactly as per [RFC3168].

3.4. Handling of SYN, SYN-ACK and RST Packets

[RFC3168] states that "A host MUST NOT set ECT on SYN or SYN-ACK packets." [RFC5562] proposes setting ECT on SYN-ACK packets, but maintains the restriction of no ECT on SYN packets. Both these RFCs prohibit ECT in SYN packets due to security concerns regarding malicious SYN packets with ECT set. These RFCs, however, are intended for general Internet use, and do not directly apply to a controlled datacenter deployment. The switching fabric can drop TCP packets that do not have the ECT set in the IP header. If SYN and SYN-ACK packets for DCTCP connections are non-ECT they will be dropped with high probability. For DCTCP connections SYN, SYN-ACK and RST packets are sent with ECT set.

4. Implementation Issues

As noted in Section 3.3, the implementation must choose a suitable estimation gain. [DCTCP10] provides a theoretical basis for selecting the gain. However, it may be more practical to use experimentation to select a suitable gain for a particular network and workload. The Microsoft implementation of DCTCP in Windows Server 2012 uses a fixed estimation gain of 1/16.

The implementation must also decide when to use DCTCP. Datacenter servers may need to communicate with endpoints outside the datacenter, where DCTCP is unsuitable or unsupported. Thus, a global configuration setting to enable DCTCP will generally not suffice. DCTCP may be configured based on the IP address of the remote endpoint. Microsoft Windows Server 2012 also supports automatic selection of DCTCP if the estimated RTT is less than 10 ms and ECN is successfully negotiated, under the assumption that if the RTT is low, then the two endpoints are likely on the same datacenter network.

5. Deployment Issues

Since DCTCP relies on congestion marking by the switch, DCTCP can only be deployed in datacenters where the network infrastructure supports ECN. The switches may also support configuration of the congestion threshold used for marking. [DCTCP10] provides a theoretical basis for selecting the congestion threshold, but as with estimation gain, it may be more practical to rely on experimentation or simply to use the default configuration of the device.

DCTCP requires changes on both the sender and the receiver, so both endpoints must support DCTCP. Furthermore, DCTCP provides no mechanism for negotiating its use, so both endpoints must be configured through some out-of-band mechanism to use DCTCP. A variant of DCTCP that can be deployed unilaterally and only requires standard ECN behavior has been described in [ODCTCP][BSDCAN], but requires additional experimental evaluation.

6. Known Issues

DCTCP relies on the sender's ability to reconstruct the stream of CE codepoints received by the remote endpoint. To accomplish this, DCTCP avoids using a single ACK packet to acknowledge segments received both with and without the CE codepoint set. However, if an ACK packet is dropped, it's possible that a subsequent ACK will indeed acknowledge a mix of CE and non-CE segments. This will, of course, result in a less accurate congestion estimate. There are some potential mitigations:

o Even with a degraded congestion estimate, DCTCP may still perform better than [RFC3168].

o If the estimation gain is small relative to the packet loss rate, the estimate may not be degraded much.

o If packet losses mostly occur under heavy congestion, most drops will occur during an unbroken string of CE packets, and the estimate will be unaffected.

However, the affect of packet drops on DCTCP under real world conditions has not been analyzed.

DCTCP provides no mechanism for negotiating its use. Thus, there is additional management and configuration overhead required to ensure that DCTCP is not used with non-DCTCP endpoints. The affect of using DCTCP with a standard ECN endpoint has been analyzed in [ODCTCP][BSDCAN]. Furthermore, it's possible that other implementations may also modify [RFC3168] behavior without negotiation, causing further interoperability issues.

Much like standard TCP, DCTCP is biased against flows with longer RTTs. A method for improving the fairness of DCTCP has been proposed in [ADCTCP], but requires additional experimental evaluation.

7. Implementation Status

This section documents the implementation status of the specification in this document, as recommended by [RFC6982].

This document describes DCTCP as implemented in Microsoft Windows Server 2012. Since publication of the first versions of this document, the Linux [LINUX] and FreeBSD [FREEBSD] operating systems have also implemented support for DCTCP in a way that is believed to follow this document.

8. Security Considerations

DCTCP enhances ECN and thus inherits the security considerations discussed in [RFC3168]. The processing changes introduced by DCTCP do not exacerbate these considerations or introduce new ones. In particular, with either algorithm, the network infrastructure or the remote endpoint can falsely report congestion and thus cause the sender to reduce cwnd. However, this is no worse than what can be achieved by simply dropping packets.

9. IANA Considerations

This document has no actions for IANA.

10. Acknowledgements

The DCTCP algorithm was originally proposed and analyzed in [DCTCP10] by Mohammad Alizadeh, Albert Greenberg, Dave Maltz, Jitu Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan.

Lars Eggert has received funding from the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 ("SSICLOPS"). This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

11. References

11.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC2018] Mathis, M., Mahdavi, J., Floyd, S., and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

11.2. Informative References

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

[RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, June 2009.

[RFC6982] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", RFC 6982, July 2013.

[DCTCP10] Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and M. Sridharan, "Data Center TCP (DCTCP)", Proc. ACM SIGCOMM 2010 Conference (SIGCOMM 10), August 2010, <http://dl.acm.org/citation.cfm?doid=1851182.1851192>.

[ODCTCP] Kato, M., "Improving Transmission Performance with One-Sided Datacenter TCP", M.S. Thesis, Keio University, 2014, <http://eggert.org/students/kato-thesis.pdf>.

[BSDCAN] Kato, M., Eggert, L., Zimmermann, A., van Meter, R., and H. Tokuda, "Extensions to FreeBSD Datacenter TCP for Incremental Deployment Support", BSDCan 2015, June 2015, <https://www.bsdcan.org/2015/schedule/events/559.en.html>.

[ADCTCP] Alizadeh, M., Javanmard, A., and B. Prabhakar, "Analysis of DCTCP: Stability, Convergence, and Fairness", Proc. ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 11), June 2011, <https://dl.acm.org/citation.cfm?id=1993753>.

[LINUX] Borkmann, D. and F. Westphal, "Linux DCTCP patch", 2014, <https://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/commit/?id=e3118e8359bb7c59555aca60c725106e6d78c5ce>.

[FREEBSD] Kato, M. and H. Panchasara, "DCTCP (Data Center TCP) implementation", 2015, <https://github.com/freebsd/freebsd/commit/8ad879445281027858a7fa706d13e458095b595f>.

[MORGANSTANLEY]
Judd, G., "Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter", Proc. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), May 2015, <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/judd>.

Authors' Addresses

Stephen Bensley
Microsoft
One Microsoft Way
Redmond, WA 98052
USA

Phone: +1 425 703 5570
Email: sbens@microsoft.com

Lars Eggert
NetApp
Sonnenallee 1
Kirchheim 85551
Germany

Phone: +49 151 120 55791
Email: lars@netapp.com
URI: http://eggert.org/

Dave Thaler
Microsoft

Phone: +1 425 703 8835
Email: dthaler@microsoft.com

Praveen Balasubramanian
Microsoft

Phone: +1 425 538 2782
Email: pravb@microsoft.com

Glenn Judd
Morgan Stanley

Phone: +1
Email: glenn.judd@morganstanley.com