Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 18, 2014

S. Bensley
Microsoft
L. Eggert
NetApp
D. Thaler
Microsoft

February 14, 2014

Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters
draft-bensley-tcpm-dctcp-00

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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 18, 2014.

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 switch's queue occupancy:

- The queue must be short enough that it doesn't impose excessive latency on short flows.

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

2. DCTCP Algorithm

There are three components involved in the DCTCP algorithm:

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

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

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

2.1. 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 the switch's queue length is greater than the congestion threshold, the switch

sets the CE codepoint in the packet. 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.

2.2. 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, we introduce 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 is set if and only if DCTCP.CE is true. When receiving packets, the CE codepoint is 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, the CE codepoint is ignored.

2.3. 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 updated as follows:

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

where

- g is the estimation gain, a real number between 0 and 1. The selection of g is left to the implementation.

- 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).

Whenever the TCP congestion estimate is updated, the sender also updates the TCP congestion window as follows:

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

Thus, when there is no congestion at all, Alpha equals zero, and the congestion window is left unchanged. When there is total congestion, Alpha equals one, and the congestion window is reduced by half. Lower levels of congestion will result in correspondingly lesser reductions to the congestion window.

In order to update DCTCP.Alpha, we make use of the TCP state variables defined in [RFC0793], and introduce three additional TCP state variables:

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

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

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

The congestion estimator on the sender processes acceptable ACKs as follows:

1. Compute the bytes acknowledged:

   ```
   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, we've reached the end of the observation window, so proceed to update the congestion estimate.

5. Compute the congestion for the current window:

M = DCTCP.BytesMarked / DCTCP.BytesSent

6. Update the congestion estimate:

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

7. Set the end of the new window:

DCTCP.WindowEnd = SND.NXT

8. Reset the byte counters:

DCTCP.BytesSent = DCTCP.BytesMarked = 0

3. Implementation Issues

As noted in Section 2.3, 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 msec, under the assumption that if the RTT is low, then the two endpoints are likely on the same datacenter network.

4. 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 device's default configuration.

DCTCP requires changes on both the sender and the receiver, so in a heterogeneous datacenter, all the endpoints should support DCTCP and should be configured to use it.

5. 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 its congestion window. However, this is no worse than what can be achieved by simply dropping packets.

6. IANA Considerations

This document has no actions for IANA.

7. 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.

8. References

8.1. Normative References

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

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

8.2. Informative References

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

[DCTCP10] Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and M. Sridharan, "Data Center TCP (DCTCP)", December 2010, <http://www.sigcomm.org/ccr/papers/2010/October/1851275.1851192/>.

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

Dave Thaler
Microsoft

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